Eindhoven University of Technology

MASTER

A hybrid level-set method on the GPU

Timmermans, L.J.L.

Award date:
2012

Link to publication

Disclaimer
This document contains a student thesis (bachelor's or master's), as authored by a student at Eindhoven University of Technology. Student theses are made available in the TU/e repository upon obtaining the required degree. The grade received is not published on the document as presented in the repository. The required complexity or quality of research of student theses may vary by program, and the required minimum study period may vary in duration.

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain

A Hybrid Level-Set Method On The GPU

Master's thesis

L.J.L Timmermans

December 2011

Supervisor: dr. A.C. Jalba
Assessment Committee: dr. M.A. Westenberg
dr. B. Mesman

Technische Universiteit Eindhoven University of Technology

ABSTRACT

Historically, there have been two approaches to tracking evolving interfaces: Lagrangian and Eulerian. While it is difficult to track an interface that changes topology using a Lagrangian approach, the Eulerian approach handles such cases with ease. Unless expensive high-order finite differencing schemes are used however, the Eulerian approach exhibits what is observed as a loss of mass. These orthogonal strengths and weaknesses of the Lagrangian and Eulerian approaches have lead to an interest in a third, hybrid approach.

A number of hybrid methods have been proposed in recent literature which augment the level-set method, which takes an Eulerian approach, with Lagrangian-style particles. In this thesis, a comparison is made between a number of these proposals. The Self-Adaptive Oriented Particles Level-Set (SA-OPLS) method was found to provide excellent results with respect to mass-preservation, while additionally containing a robust self-adaptive mechanism.

The advent of programmable graphics processing units (GPUs) has provided access to a highly parallel computing platform. Both the Lagrangian and Eulerian approach, and therefore naturally also a hybrid approach, have been recognized to exhibit a large amount of parallelism, allowing them to benefit from such highly parallel architectures. One example is the GPU-accelerated Sorted Tile List (GPU-STL) method, which provides a highly efficient implementation of the level-set method.

In this thesis, a hybrid level-set implementation on the GPU is proposed, based on the combination of GPU-STL and SA-OPLS. The final results of this implementation are promising, comparing favourably against similar work presented in recent related literature.

CONTENTS

1 INTRODUCTION 1
1.1 Overview . . . 1
1.2 Numerical methods for tracking an evolving interface . . . 2
1.2.1 The Lagrangian point of view. . . . 3
1.2.2 The Eulerian point of view. . . . 4
1.2.3 A hybrid point of view . . . 8
2 RELATED WORK 9
2.1 Hybrid methods . . . 9
2.2 GPU-accelerated implementations . . . 10
3 COMPARISON OF MASS-PRESERVING LEVELSET METHODS 13
3.1 Benchmark tests . . . 13
3.2 Benchmark results . . . 14
3.3 Particle addition/deletion . . . 16
3.4 Choosing a method. . . . 18
4 THE SELF-ADAPTIVE ORIENTED PARTICLES LEVEL-SET METHOD 25
4.1 Correction . . . 25
4.2 Particle advection . . . 26
4.3 Self-adaptiveness . . . 27
4.3.1 Particle addition . . . 27
4.3.2 Particle deletion . . . 30
5 BACKGROUND ON GPUS AND THE STL METHOD 31
5.1 The GPU platform and CUDA framework . . . 31
5.1.1 GPUs . . . 31
5.1.2 CUDA . . . 31
5.2 The Sorted Tile List method . . . 33
6 ALGORITHM DESIGN AND IMPLEMENTATION OF THE HYBRID METHOD USING CUDA 35
6.1 Particle Advection . . . 35
6.2 Level-set Correction . . . 35
6.3 Particle Management . . . 35
6.4 Render Particles . . . 35
7 RESULTS 37
7.1 Test setup . . . 37
7.2 The Enright test . . . 37
7.3 Particle initialization . . . 38
7.4 Particle Advection . . . 39
7.4.1 Without Self-adaptiveness . . . 40
7.4.2 With Self-adaptiveness . . . 43
7.5 Level-set Correction . . . 45
7.6 Particle Management . . . 46
7.7 Overview . . . 48
7.8 Comparison to related work . . . 50
8 CONCLUSIONS AND FUTURE WORK 53

BIBLIOGRAPHY 55

1

INTRODUCTION

There are many interesting problems that involve tracking or capturing an evolving interface that separates two regions from each other. Perhaps the most straightforward examples are in simulations of physical phenomena, where the interface might represent a dynamic boundary between different materials. Evolving interfaces are also often found in computer graphics, for example in computer-generated special effects of fluids used in Hollywood movies. There is also a wide range of problems that can be translated such that they involve dynamically-evolving interfaces. One example is image segmentation, which is a task found often in computer vision and image processing. In most cases these interfaces are represented by a closed curve in 2D or a closed surface in 3D. Figure 1 gives an impression of some of these problems that involve interface tracking.

1.1 OVERVIEW

There are two main challenges in finding solutions to these types of interface tracking problems:

- A domain or problem-specific challenge of finding a mathematical description, in the form of a differential equation, that describes how the interface should evolve. In the example of a fluid simulation these could be the Navier-Stokes equations that describe the laws of physics that apply to the motion of fluids. Since we are interested in the position of the interface as it evolves over time, these differential equations describe a velocity. As mentioned in [29], such a velocity function can depend on three factors: local properties of the interface such as curvature, global properties such as its length or area and independent properties such as time and space.
- The challenge of finding a mathematical description or model for the interface that can be used to track its evolution according to a velocity function as described above. These are numerical methods and, as they are solved in a digital domain, one of the key issues lies in discretizing the spatial and temporal domains.

Because the latter is not a problem-specific issue, there has been great interest in finding a generically applicable method. As described in [28], such a method might meet the following challenging issues:

- interfaces that change topology, break, and merge as they move;
- formation of sharp corners, cusps, and singularities;
- dependence of the interface motion on delicate geometric quantities such as curvature and normal direction;
- complexities in three dimensions and higher;
- subtle feedback between the physics and chemistry off the interface and the position and motion of the front itself.

Various methods have been proposed over time which are commonly classified as being either Lagrangian or Eulerian approaches. Interestingly, these two approaches often seem to have orthogonal strengths and weaknesses. This is why there has been an interest in trying to combine them, leading to a third class of so-called hybrid methods.

Current methods are able to achieve high accuracy but at the cost of being computationally intensive. This limits their practical use, especially in real-time or highly interactive applications. In a recent

(a) Ion milling: experiment (top) vs. simulation (bottom).[29]

(b) Simulation and visualization of waves breaking.[17]

(c) Medical image segmentation.[14]

Figure 1: Examples of interface tracking problems.

paper, van der Laan et al.[34] introduce a memory and computation efficient approach for the Level Set Method, a Eulerian-type method, which they call the *Sorted Tile List* (STL) method. Perhaps more importantly the authors show that their STL method can greatly benefit from parallelization. With the advent of programmable *graphics processing units* (GPUs) found on commodity graphics hardware, there is now a tool with which this parallelism can be exploited. In a yet to be published follow-up paper, an outline of such an implementation and results showing the efficiency of this GPU-STL method are given.

This brings us to the goal of this thesis, which is to *improve on the GPU-STL method by adapting it to become a hybrid method.* This results in the following contributions:

- A comparison of a number of different hybrid methods proposed in recent literature.
- A mapping of algorithms needed for the selected hybrid method to a GPU architecture.
- A thorough analysis of results obtained from the gpu-accelerated implementation of the selected hybrid method.

The structure of this thesis is as follows. The following section describes the aforementioned Eulerian, Lagrangian and hybrid approaches in more detail. Related work on hybrid and GPU accelerated implementations is then described in Chapter 2. In Chapter 3, a comparison is made between a number of different proposals for a hybrid method found in related work. Based on the results of this comparison, the method that is found to be the most suitable for implementation on top of the GPU-STL method is selected and discussed in more detail in Chapter 4. Chapter 5 then provides background information on GPU programming and the STL method. The actual implementation is then described in Chapter 6 and results are shown in Chapter 7. Finally, Chapter 8 presents the conclusions and discusses possible future work.

1.2 NUMERICAL METHODS FOR TRACKING AN EVOLVING INTERFACE

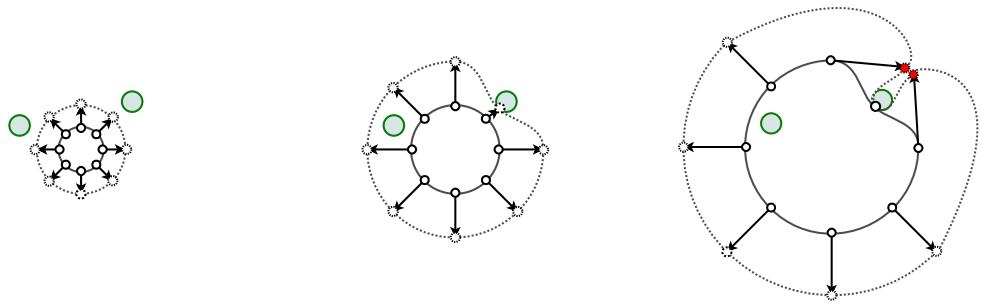
As noted in Section 1.1, numerical methods used for tracking an interface have been commonly classified as being either Lagrangian or Eulerian. The difference between these two classes is in the way the interface is 'viewed'. In the Lagrangian approach, the interface is viewed from points on the interface itself. Eulerian approaches view the interface from fixed positions as it 'passes by'. The following sections describe these two approaches and their difficulties. For a more in-depth discussion the reader is referred to [29] or [25].

1.2.1 *The Lagrangian point of view.*

In Lagrangian methods the interface is discretized into a set of particles, sometimes also called markers, that are positioned *on the interface* and that know their direct neighbours. To evolve the interface, the positions of the particles are integrated over time using a given velocity function $\mathbf{u}$. Let $\mathbf{x}^P$ be the position of a particle P, then the evolution equation for its position is given by

$$\frac{d\mathbf{x}^P}{dt} = \mathbf{u} \tag{1.1}$$

At each point in time, the particles can then be used to reconstruct the interface by fitting a curve through these particles.



(a) Initially-placed interface evolves in the direction of the normal.

(b) The object on the left is not captured at all.

(c) The object on the right will also not be captured without any mechanism to change topology, i.e connecting the two red particles creating a separate interface.

Figure 2: Example of an interface tracking problem inspired by image segmentation. Particles located on the interface move in the direction of the normal (depicted by the arrows), thus evolving the interface (shown as a dashed line). The goal is for the interface to capture each of the objects shown in green independently.

An example of a front tracking problem inspired by image segmentation is given in Figure 2. Figure 2a shows an initially-placed interface as a closed curve described by particles. The objective is to let the interface evolve in such a way that it captures the two green-colored objects. In order to achieve this, the velocity-function is set to a constant in the normal direction, allowing the interface to expand as indicated by the arrows. At the positions occupied by the objects the velocity-function is zero, effectively trapping the particles and therefore the interface there.

Figure 2 also illustrates some of the difficulties faced by Lagrangian methods:

- *Particle density*
  As the interface evolves, its area or length is most likely to change. If the number of particles used to discretize the interface is kept constant, then the density of particles on the interface will also change. This can be seen in Figure 2 where the length of the interface increases as it expands, therefore decreasing particle density. One possible effect of a low particle density can be observed in Figure 2b where the object on the left is not being captured by the interface as none of the particles encounter it.

  A high density, on the other hand, can also be undesirable. With particles close to each other, a feedback loop that causes instability can occur. Small and unavoidable errors in the positions of the particles lead to local variations in the computed spatial derivatives. If the velocity function depends on local properties, such as curvature, that rely on these derivatives then these variations propagate through the velocity function. This causes the variations in spatial derivatives to grow uncontrollably. Only with a small enough timestep for integration of the particle positions can this be prevented. Such small timesteps can however render the method impractical.

In order to deal with these problems, Lagrangian methods often need to apply a form of particle addition/deletion or redistribution. Other solutions include filtering or smoothing techniques that prevent small variations in the spatial derivatives.

- *Topological changes and self-intersection*
  In order for the object in Figure 2c to be properly captured it is necessary for the two particles indicated in red to disconnect from their current neighbour and to become each others neighbour. These particles then together form a separate interface capturing the object. In other words: a change in topology is needed. It is difficult to find a systematic solution to detect and handle these topological changes, especially in higher-dimensional problems.

  There is nothing preventing the red particles in Figure 2c from crossing each other as time evolves. This leads to a self-intersecting interface, and while this might be valid mathematically for a Lagrangian approach, in the context of a moving boundary separating two regions this is generally not be desirable. Take the example of a flame incinerating a piece of paper, in which the interface is the boundary separating the burnt from the unburnt regions. If the interface self-intersects this would indicate a burnt region being burned again. As with topological changes, complex methods for eliminating and re-organizing particles are needed to solve this problem.

1.2.2 *The Eulerian point of view.*

Although it is certainly not the only method employing an Eulerian approach, the Level Set Method introduced by Osher and Sethian in [26] has become one of the most popular techniques for capturing and tracking evolving interfaces. In most Eulerian approaches, a new unknown function is introduced that implicitly represents the interface on a fixed grid. In the Level Set Method this unknown function is the scalar function ϕ, and an interface Γ is represented by the level set in which this function has a given constant value. Thus, the interface can be seen as being the level curve or level surface (depending on dimensionality) of ϕ, hence the name Level Set Method. More formally, the closed interface Γ is defined as

$$\Gamma = \{(\mathbf{x}, t) | \phi(\mathbf{x}, t) = c\}, \tag{1.2}$$

where $\mathbf{x}$ is a position, t denotes time and c is a constant usually taken to be $c = 0$.

To further illustrate this, Figure 3a shows two circular closed interfaces that separate the blue-colored interior regions from the exterior region. These interfaces are the zero level sets, or iso-contours, of the function ϕ, which is represented by the red cones. Assuming these interfaces expand, evolving them then becomes equivalent to evolving the function ϕ through time, moving it upward as shown in Figure 3b. Note also that with the interface being the zero iso-contour, the interior and exterior regions are easily identified by observing the sign of ϕ.

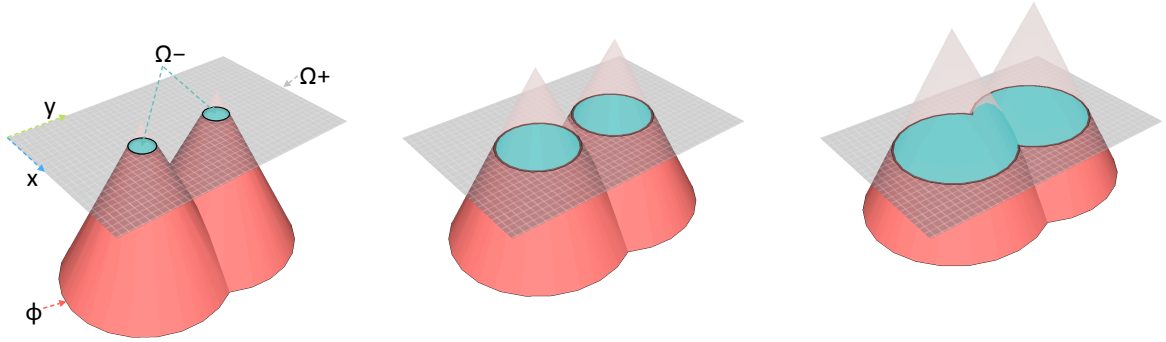
Evolving ϕ even further leads to the result shown in Figure 3c. As the interfaces expanded, a topological change has taken place, joining the two regions and forming a single interface. The ease with which this topological change has been handled is one of the greatest advantages of an Eulerian approach over the Lagrangian approach described in Section 1.2.1.

It is desirable for ϕ to be a Lipschitz smooth function in order to enable reliable differentiation. A signed distance function has been found to be a good candidate for the scalar function ϕ. A distance function $d(\mathbf{x})$ is defined as

$$d(\mathbf{x}) = \min_{I}(\|\mathbf{x} - \mathbf{x_I}\|), \tag{1.3}$$

where $\mathbf{x_I}$ is a position on the interface and the $\|\cdot\|$ operator returns the Euclidean norm, or modulus, of a vector. This function can be seen as returning the distance from a point $\mathbf{x}$ to a point $\mathbf{x_I}$ on the interface that is closest to it. Let Ω^- be the interior and Ω^+ the exterior region with respect to the closed interface Γ, then ϕ is defined to be a signed distance function as follows

$$\phi(\mathbf{x}) = \begin{cases} -d(\mathbf{x}) & \text{if } \mathbf{x} \in \Omega^- \\ d(\mathbf{x}) & \text{if } \mathbf{x} \in \Omega^+ \end{cases} \tag{1.4}$$



(a) (b) (c)

Figure 3: Example of two expanding circular interfaces using the Level Set Method. The Level set function ϕ is colored red and a cross-section is used to indicate the level set. The interior and exterior regions which are separated by the interface are denoted by Ω^- and Ω^+ respectively. As ϕ evolves through time (b), the two circular interfaces merge (c).

With ϕ being a signed distance function, it has the interesting property that

$$\|\nabla\phi\| = 1, \tag{1.5}$$

where ∇ is the gradient operator that returns a vector which has partial derivatives as its components, i.e.

$$\nabla\phi(\mathbf{x}) = \left(\frac{\partial\phi}{\partial x_0}, \cdots, \frac{\partial\phi}{\partial x_n}\right) \tag{1.6}$$

Intuitively the property in Equation 1.5 expresses that moving twice as close towards the interface will halve the value of ϕ which makes sense when ϕ expresses distance. This property is however only true in a general sense. It is not satisfied for any point that is equidistant from at least two other points on the interface such as the center of a circular interface for example. This means that it is possible that locally $\|\nabla\phi\| \neq 1$. Fortunately this generally does not occur near the interface and the effects are smeared out by discretization.

As the gradient of ϕ results in a vector pointing in the direction of the greatest rate of increase, it can be used to find the outward unit normal when evaluated at a point $\mathbf{x_I}$ on the interface:

$$\mathbf{N}_{x_I} = \frac{\nabla\phi_{x_I}}{\|\nabla\phi_{x_I}\|} \tag{1.7}$$

Using Equation 1.5, the evaluation of the outward unit normal can be simplified by taking out the normalization:

$$\mathbf{N}_{x_I} = \nabla\phi_{x_I} \tag{1.8}$$

Given a velocity u which describes the velocity of the interface in its normal direction and Equation 1.8, we can now construct a scalar transport equation for the time evolution of ϕ:

$$\frac{\partial\phi}{\partial t} + u\|\nabla\phi\| = 0 \tag{1.9}$$

This partial differential equation, commonly known as the *level set equation*, is solved numerically on a grid. Alternatively, the velocity function might be given by a vector field $\mathbf{u}$ in which case the transport equation becomes

$$\frac{\partial\phi}{\partial t} + \mathbf{u}\cdot\nabla\phi = 0 \tag{1.10}$$

1.83	1.24	1.00	1.24	1.83
1.24	0.41	0.00	0.41	1.24
1.00	0.00	-1.00	0.00	1.00
1.24	0.41	0.00	0.41	1.24
1.83	1.24	1.00	1.24	1.83

y x

(a) Initially placed circular interface (at $t = 0$)

2.22	1.53	1.12	1.12	1.53
1.70	0.83	0.21	0.21	0.83
1.50	0.50	-0.50	-0.50	0.50
1.70	0.83	0.21	0.21	0.83
2.22	1.53	1.12	1.12	1.53

y x

(b) circular interface after one timestep (at $t = 0.5$)

Figure 4: Example of an interface implicitly described as the zero level set of ϕ (discretized on a grid) and its time evolution.

To illustrate how ϕ is discretized and evolved using Equation 1.10, an example is given in Figure 4. In Figure 4a the signed distance function $\phi(x, y, t)$ is initialised for $t = 0$ with its circular zero level set indicated in grey. Using a simple velocity function to translate the interface horizontally to the right, $\mathbf{u} = \binom{1}{0}$, we solve Equation 1.10 for all points on the grid with a timestep of $\Delta t = 0.5$. For time discretization, a simple first-order accurate forward Euler method is used, allowing us to write:

$$\frac{\phi(t + \Delta t) - \phi(t)}{\Delta t} + \mathbf{u}(t) \cdot \nabla\phi(t) = 0 \tag{1.11}$$

For spatial discretization of $\nabla\phi$ a little more care needs to be taken. In our example the information contained by ϕ moves from left to right as dictated by the velocity function. It is therefore appropriate to use information from the left of a position on the grid to approximate the partial derivatives of $\nabla\phi$. Therefore, an appropriate first-order accurate scheme for this case is the backward differencing scheme given in Equation 1.12. As the direction of $\mathbf{u}$ can vary in both time and space, a more general method is needed that selects the appropriate differencing scheme based on the local direction of $\mathbf{u}$. These methods are called upwinding methods and can be described as

$$D^{-i}(\mathbf{x}) = \frac{\phi(\mathbf{x}) - \phi(\mathbf{x} - \Delta i)}{\Delta i} \tag{1.12}$$

$$D^{+i}(\mathbf{x}) = \frac{\phi(\mathbf{x} + \Delta i) - \phi(\mathbf{x})}{\Delta i} \tag{1.13}$$

$$\frac{\partial\phi}{\partial i}(\mathbf{x}) = \max(u_i(\mathbf{x}), 0)D^{-i}(\mathbf{x}) + \min(u_i(\mathbf{x}), 0)D^{+i}(\mathbf{x}) \tag{1.14}$$

where i is a component of $\mathbf{x}$ and Equation 1.14 is the resulting upwind derivative evaluated at $\mathbf{x}$.

To guarantee stability with this scheme, the Courant-Friedreichs-Lewy condition (CFL-condition) needs to be enforced. The CFL-condition, given in Equation 1.15 restricts the time-step in order to prevent the physical wave, modeled by the velocity function, from propagating faster than the numerical wave, discretized on the grid.

$$\Delta t \cdot \max\left(\Sigma_{i=0}^{n} \frac{|u_i|}{\Delta i}\right) < \alpha \qquad \text{where} \qquad 0 < \alpha < 1 \tag{1.15}$$

Although all examples given above are of interfaces in two dimensions, it requires very little effort to extend the method to three or more dimensions. Visualization of the implicit interface can be achieved by well-known algorithms such as marching squares/cubes. As shown before, no complex methods are required to deal with topological changes. This makes the Level Set Method as described above an elegant and powerful technique that is relatively easy to implement.

There are however also a number of issues to be dealt with in the Level Set Method:

- *The method is computationally intensive.*
  Without any optimisations, the entire grid needs to be stored and updated at every time-step of

the evolution. In most practical cases these grids are square and 2 or 3-dimensional. This results in a time complexity that is polynomial in the resolution of the grid. Especially in three dimensions this can quickly lead to impractical scenarios, in which resolution needs to be sacrificed.

One optimisation comes from the observation that only grid points within a band near the interface are actually needed for describing and evolving the interface. In such an approach, computational intensity only depends on the width of this band and the length or area of the interface, thus greatly reducing average computational intensity. The resulting method is known as the narrow band method [4].

Apart from optimizations such as the narrow band method, there has been a growing interest in exploiting the computational power of parallel processing architectures for level set methods. A number of implementations have been proposed that employ GPUs in order to benefit from the parallel nature of the level set method. Section 2.2 will provide an overview of related work on GPU implementations.

- *The method has high memory requirements.*
  Storing a full grid also leads to high memory requirements. This is especially problematic on GPUs, which still have a limited amount of memory. By using dynamic tree-like structures that only describe the grid near the interface at its full resolution, the average memory requirements can be decreased. Taking this one step further by only storing the grid in a narrow-band, are sparse methods such as DT-Grid[24] and the aforementioned STL. As with computational complexity, the memory requirements of these methods then depend on the length or area of the interface.

- *Maintaining ϕ to be a signed distance function.*
  As the level set function ϕ is evolved through time, it is possible that the velocity function causes the level sets to converge or diverge. This results in ϕ no longer being a proper signed distance function which leads to inaccurate solutions of the level set equation.

  One way of forcing ϕ to be a signed distance function is by periodic re-initialisation. In [32] the authors describe how to (re)initialise ϕ without explicitly finding its zero level set. This is done by solving Equation 1.16 to steady state.

$$\frac{\partial \phi}{\partial t} = \mathrm{sign}(\phi_0)(1 - \|\nabla \phi\|) \tag{1.16}$$

  Taking a smoother approximation for $\mathrm{sign}(\phi_0)$, such as Equation 1.17, has been found to lead to better results[25, p.67].

$$\mathrm{sign}(\phi_0) = \frac{\phi_0}{\sqrt{\phi_0^2 + \|\nabla \phi\|}} \tag{1.17}$$

  Periodically solving Equation 1.16 for a couple of iterations during the time evolution of ϕ forces it to stay close to a signed distance function. The disadvantage of this approach is that it can lead to a displacement of the zero level set.

  Another solution is to add an additional term to the level set equation (Equation 1.10) as proposed in [15]. This term is called a rescaling speed term, and one possible term is the right-hand side of Equation 1.16.

- *Loss of mass due to numerical diffusion*
  The Eulerian nature of the Level Set Method with its finite differencing to approximate gradients leads to errors that are commonly referred to as numerical diffusion. The effects of numerical diffusion can be profound and are observable as a 'loss of mass'. The solution to this is to use higher-order temporal and spatial discretization-schemes that, due to their higher accuracy, suffer less from numerical diffusion. Modern implementations, such as DT-Grid[24], use 3rd order Runge-Kutta scheme for time and a 5th order HJ-WENO[13] scheme for space. Unfortunately, these schemes increase the computational burden of solving the level set equation.

1.2.3 *A hybrid point of view*

From the above discussion of Lagrangian and Eulerian approaches one might notice that they share some orthogonal strengths and weaknesses. A Lagrangian description can accurately track discrete positions on the interface at relatively low computational costs. It has problems however with accurately constructing the interface and handling topological changes. The Eulerian Level Set Method on the other hand is able to maintain a nice representation of the interface and handles topological changes with ease, but comes at a high computational cost and exhibits loss of mass due to numerical diffusion.

The observation that Lagrangian and Eulerian approaches share orthogonal strengths and weaknesses has led to a number of proposals for a hybrid approach. The global idea of such a hybrid approach is that an Eulerian method, such as the Level Set method, is used to represent and evolve the interface while Lagrangian markers (that are evolved independently) are used to perform local corrections of the level set. The resulting method can still handle topological changes, but is remarkably effective in preserving mass, often even removing the need for the expensive higher-order RK and HJ-WENO schemes for level set evolution.

The *Particle Level Set Method* (PLS) proposed by Enright et al. in [8] was the first such hybrid method, and it remains popular even today. There have been a number of other proposals however that mainly differ in the way the level set correction is performed. In the next chapter, an overview of these proposed hybrid methods is given.

2

RELATED WORK

We start this overview of related work with a short discussion of a few methods that are aimed at creating an efficient implementation of the Level Set Method. As mentioned in Section 1.2.2, there have been a number of proposals for combining Eulerian and Lagrangian approaches. An overview of these is given in Section 2.1 and a comparison will be made in Chapter 3 in order to select a suitable method for our implementation. Section 2.2 discusses related work on GPU-accelerated implementations.

The *Fast Marching Method* (FMM) as described by Sethian in [29] is closely related to Dijkstra's famous shortest path finding algorithm, and is based on the assumption that information always flows in the same direction, i.e. the velocity-function does not change its sign. FMM is frequently used for example for efficiently performing the level set reinitialisation described in Section 1.2.2. Unfortunately, besides limiting the scope of interface-problems that can be solved, FMM relies on a heap data structure making an efficient implementation on a GPU difficult.

As mentioned in Section 1.2.2, in order to guarantee numerical stability when solving the level set equation there is a bound on the timestep given by the CFL-condition. This bound can be very restrictive which makes evolution of the interface computationally intensive. By employing a *semi-lagrangian* advection scheme, as proposed in [30], the evolution becomes unconditionally stable thus removing the restrictive CFL condition. Unfortunately this also limits the class of problems to hyperbolic ones.

One of the first methods to maintain a sparse representation of ϕ was the *Dynamic Tubular Grid* (DT-Grid) method by Nielsen and Museth. In [24], the authors propose data structures for this sparse grid including algorithms with which these structures can be accessed and traversed. This results in a method that is not only memory-efficient but, according to the authors, also outperforms (at that time) existing narrow-band methods.

The *Sorted Tile List* (STL) approach described by van der Laan et al. in [34] is aimed at providing a memory and computationally efficient implementation for the Level Set Method much like DT-Grid. The main idea of the STL method is to represent the narrow band around the interface as a list of fixed-size blocks called tiles. By keeping this list of tiles lexicographically sorted it is possible to efficiently access and traverse the tiles. The authors show that their method compares favorably against other current implementations such as DT-Grid. Perhaps more interestingly, they show that their STL method can greatly benefit from both fine and course-grain parallelization.

2.1 HYBRID METHODS

The *Particle Level Set* (PLS) method briefly mentioned in the previous chapter and introduced by Enright et al. in [8] is an extension of the work done on hybridizing Eulerian and Lagrangian methods by Foster and Fedkiw in [10]. The idea of the PLS method is to place fictitious particles within a band on both sides of the interface that is tracked by the level set method. These particles are given a sign which corresponds to the sign of the level set function at the particle's position. The particles and level set function are then independently evolved through time. Enright et al. use a 5th order accurate HJ-WENO scheme for spatial derivatives of the level set function and 3rd order TVD Runge-Kutta for time integration of both the level set PDE (Equation 1.10) and particle positions. After each time step, the level set function is locally corrected in those places in which a particle is said to have 'escaped'. Escaped particles are particles that have a sign that contradicts with the information expressed by the level set function, in other words: they are found on the wrong side of the interface. The authors also note that as the interface stretches, the density of particles near the interface can become low thus making accurate correction of the level set in those places difficult. The proposed solution is to

periodically reseed particles wherever the density has become low. It is noted however that newly-added particles might represent less accurate information and as a result their correction of the level set can damage rather than improve the resolution of the level set. In [9], Enright et al. demonstrate that the excellent corrective capabilities of the particles alleviate the need for computationally-expensive high order schemes, such as 5th order HJ-WENO. They propose a more efficient method using a first order semi-lagrangian method for level set evolution combined with a Fast Marching method for reinitialisation and a second order Runge-Kutta midpoint scheme for particle evolution.

In the *Marker Level Set* (MLS) method described in [21, 22], fictitious particles, similar to those used in PLS, called markers are placed on the interface itself rather than in a band around it as in the PLS method. As such, the MLS method needs less markers while, according to the authors, obtaining similar or even better mass-preserving results when compared to the PLS method. Since the particles are known to be positioned on the interface, a correction of the level set at a certain location can be performed by determining the distance to the particle closest to it within a certain neighbourhood. Alternatively, as described in [21], one might consider using a weighted average of all particles within a certain neighbourhood. A relatively simple mechanism is proposed for marker addition and deletion which uses a threshold on the density of markers inside a cell. New markers are positioned on the interface using a local linear reconstruction of the interface based on the level set.

In a poster-presentation[27], Pighin et al. describe that, by adding information on a particle's orientation with respect to the interface, it is possible to obtain a better local estimate of the interface. The authors show that this can then be used to improve the accuracy of the correction of the level set. To the best of our knowledge, the only work published on this method is the aforementioned poster which unfortunately does not contain a full discussion of the method.

The idea of including information about orientation in the correction procedure is also discussed in [35], leading to what what the authors claim to be an improved correction procedure for the PLS method.

In [11], Ianniello and Mascio propose a *Self-Adaptive Oriented Particles Level Set* (SA-OPLS) method that is largely based on the same idea explored in the above mentioned poster-presentation[27]. The main difference is that Ianniello and Mascio accurately track the orientation of a particle by deriving an evolution equation for its normal vector. Additionally, they propose a self-adapting mechanism that can add markers in areas where their density becomes too low and remove them wherever the density becomes too high. By tracking additional local geometrical properties, the method is able to detect when to add new particles and can use a second-order local approximation of the interface in order to position these new particles.

2.2 GPU-ACCELERATED IMPLEMENTATIONS

One of the first implementations of a level set method that exploited the parallel computing capabilities of graphics hardware was due to Strzodka and Rumpf[31]. As graphics hardware evolved so did its programmability. Before the advent of general purpose GPU programming frameworks such as OpenCL, Nvidia's CUDA and Microsoft's DirectCompute, early implementations such as [31] used graphics APIs to tap into the processing power of the GPU. Many of the GPU-accelerated level set methods that have since been proposed are aimed toward image segmentation, in which mass loss is not really an issue and hybrid methods have little benefit.

Two GPU-accelerated implementations of the PLS method have been simultaneously but independently proposed by Cuntz et al.[6][7] and Matsuda et al.[18]. Both implementations use graphics APIs and shaders and show comparable resulting speedups of roughly one order of magnitude compared to CPU implementations. In a more recent poster-presentation[37], Zheng and Enright give a brief description and show results of an implementation of the PLS method using NVIDIA's CUDA. Results of their tests show a speedup of over 45x compared to a CPU implementation. To the best of our knowledge, all above mentioned GPU-PLS implementations require a full grid to be stored.

An implementation of the MLS method using graphics APIs and shaders is proposed in [19]. It, similar to the GPU-PLS implementations, also requires a full grid to be stored. The performance-results are reported to be comparable to those of the GPU-PLS implementation by Cuntz et al..

A GPU-accelerated implementation of the STL method by Jalba et al. is described in [12]. The results presented show how the STL approach indeed benefits from parallelization using NVIDIA's

CUDA framework. Because the GPU-STL method addresses both computational as well as memory requirements, it is a suitable starting-point for a hybrid implementation. The goal of this thesis is therefore to hybridize the GPU-STL method by extending it with one of the approaches described in Section 2.1.

In the next chapter, a comparison is given of the hybrid approaches listed in Section 2.1 with respect to their mass-preserving properties. Additionally, self-adaptiveness and suitability for an implementation on the GPU are discussed.

3

COMPARISON OF MASS-PRESERVING LEVELSET METHODS

This chapter provides a comparison of the following hybrid methods that have been briefly discussed in Section 2.1:

- OPLS *(Oriented Particle Level Set)* as described in [27]
- PLS *(Particle Level Set)* as described in [8]
- MLS *(Marker Level Set)* as described in [22]
- SA-OPLS *(Self-Adaptive Oriented Particle Level Set)* as described in [11]

The PLS implementation by Mokberi and Faloutsos[23] is taken as a basis and is extended with our own implementations of the remaining methods listed above. These implementations are then benchmarked for mass-preservation (not performance) using the tests described in Section 3.1. Note that all implementations and tests are 2-dimensional as this makes interpretation of the results easier. In Section 3.2 the results of these tests are reported, and a final choice is motivated in Section 3.4. In order to simplify terminology, we shall from hereon refer to markers as particles.

3.1 BENCHMARK TESTS

The mass-preserving capabilities of the hybrid methods under consideration are evaluated through two commonly used benchmark tests. Both of these tests evolve an interface in a periodic velocity-field. This allows a comparison of the area enclosed by the interface at the start and end of the period. The area enclosed by the interface is computed using a marching-squares type of algorithm with linear approximation of the interface inside a cell. All tests are performed using a 100^2 computational grid. Obviously, the area enclosed by the interface could be preserved even when this interface drastically changes shape or position. For this reason, visualizations of the results will also be presented.

The first test is a rigid body rotation of a slotted disk known as Zalesak's disk[33]. The disk (having a radius of 15 cells, slot width of 5 cells and slot length of 25 cells) is centered at $(50, 75)$. A constant vorticity velocity field given by

$$\begin{aligned} u(x,y) &= (\pi/314)(50-y) \\ v(x,y) &= (\pi/314)(x-50), \end{aligned} \tag{3.1}$$

causes the disk to rotate around the center of the computational domain with a period of 628 time units.

The second test, referred to as the Vortex-in-a-box test (derived from [5]), is a little more demanding as it stretches the interface and creates thin filaments. In this test, a circle having a radius of 15 cells, centered at $(50, 75)$, is advected using the velocity field defined by

$$\begin{aligned} u(x,y,t) &= 2\sin\left(\frac{\pi}{100}x\right)^2 \sin\left(\frac{\pi}{100}y\right)\cos\left(\frac{\pi}{100}y\right)\cos\left(\frac{\pi t}{\tau}\right) \\ v(x,y,t) &= -2\sin\left(\frac{\pi}{100}y\right)^2 \sin\left(\frac{\pi}{100}x\right)\cos\left(\frac{\pi}{100}x\right)\cos\left(\frac{\pi t}{\tau}\right), \end{aligned} \tag{3.2}$$

(derived from the stream function given in [5]), where t denotes time and τ is the period. For this test we take $\tau = 800$, such that the interface returns to its initial position after 800 time units.

For all methods under consideration we use simple (but fast) first-order methods for the advection of the levelset. More specifically, a forward Euler step as described in Equation 1.11 is used for time-integration. Spatial derivatives are obtained through the upwinding method given in Equation 1.14. The level set function is forced to stay a signed distance function through the use of the additional rescaling speed term as described in Section 1.2.2 (using the right-hand side of Equation 1.16). Results of tests of the first-order method described above for pure levelset advection without any correction by particles are also included. Furthermore, a higher-order HJ-WENO5/RK3 approach without particles, as used in the DT-grid method[24] for example, is included for reference.

For the advection of Lagrangian particles, a second-order time variation diminishing (TVD) Runge-Kutta method as described in [25] is used for time-integration. Let $\mathbf{x}^P(t)$ be the position of a particle P at time t, then $\mathbf{x}^P(t+\Delta t)$ is computed by

$$\begin{aligned} \mathbf{K_1} &= \mathbf{x}^P(t) + \Delta t\ \mathbf{u}(\mathbf{x}^P(t), t), \\ \mathbf{K_2} &= \mathbf{K_1} + \Delta t\ \mathbf{u}(\mathbf{K_1}, t+\Delta t), \\ \mathbf{x}^P(t+\Delta t) &= \frac{1}{2}(\mathbf{x}^P(t) + \mathbf{K_2}), \end{aligned} \tag{3.3}$$

where $\mathbf{u}(\mathbf{x}, t)$ is the velocity at position $\mathbf{x}$ at time t.

Particle placement, also referred to as particle seeding, can be influenced by a number of parameters. For the PLS and OPLS methods, P_{count} controls the number of particles that a cell within a band B from the interface should contain. Furthermore, cells that intersect with the interface can be made to contain a multiple of P_{count} through I_{mul}. For the MLS and SA-OPLS methods, P_{count} controls the number of particles in a cell that intersects with the interface. The placement of particles on the interface inside such a cell is based on a linear approximation of the interface inside this cell. The L_{thresh} parameter can be used to obtain a more uniform distribution of particles by only seeding particles inside cells in which the length of the interface is greater than the threshold. Initially, the number of particles is kept constant in these tests, i.e. *particle addition/deletion or self-adaptiveness is not performed*.

For the two 2-dimensional test cases under consideration, substitution in Equation 1.15 (and taking $\Delta x = \Delta y = 1$) leads to the following CFL-condition for stability:

$$\Delta t \cdot max(|u| + |v|) < \alpha \qquad where \qquad 0 < \alpha < 1$$

Taking $\Delta t = 0.2$ satisfies this condition for both tests:

- For Zalesak's Disk, $max(|u| + |v|) = |u(0,0)| + |v(0,0)| \approx 1$, thus $\alpha \approx 0.2$
- For Vortex-in-a-box, $max(|u| + |v|) \approx |u\left(\frac{100}{3}, \frac{100}{3}, 0\right)| + |v\left(\frac{100}{3}, \frac{100}{3}, 0\right)| \approx 1.3$, thus $\alpha \approx 0.26$

3.2 BENCHMARK RESULTS

Table 1 and Table 2 present results with respect to mass-preservation for Zalesak's Disk and Vortex-in-a-box tests respectively. For the sake of completeness, all relevant configuration parameters are listed in these tables, but for their definitions we refer to the relevant papers in [27], [22] and [11]. Visualizations belonging to the results are given in Figure 6 and Figure 7 respectively. Finally, results of two methods with particle addition and deletion for the Vortex-in-a-box test are presented in Table 3 and Figure 9.

The results of Zalesak's Disk test presented in Table 1 show how, without any particle correction, the simple first-order Upwind/Euler scheme suffers from 100% loss of area. The same scheme, when coupled with any of the particle correction methods, however leads to significantly better results. The higher-order HJ-WENO5/RK3 scheme also greatly reduces area-loss, but does not maintain its original shape in high-curvature regions as can be seen in Figure 6a. The corrective capabilities of the hybrid methods become even more evident in the Vortex-in-a-box test. Figure 7a shows that the HJ-WENO5/RK3 implementation is unable to maintain the fine filament of the spiral's tail. This eventually leads to almost 50% loss of area as shown in Figure 7b and Table 2.

Another result that stands out in the tests is that of the OPLS method. Its performance in terms of area-preservation, although not bad when compared to HJ-WENO/RK3, is not as good as that of the other correction methods. Far worse however, is its ability to maintain shape, as Figure 6b, Figure 7c

and Figure 7d show. It might be possible to improve these results by adjusting the weighting-function that is used in its correction-step. Unfortunately, the OPLS method as described in [27] suffers from a more structural problem. To understand this, we first explain its correction-method in a little more detail.

In the OPLS method, each particle P is characterised by its sign, denoted by $\mathrm{sign}(P)$, its position $\mathbf{x}^P$ and an orientation vector $\mathbf{d}^P$. The vector $\mathbf{d}^P$ is defined as the vector that points from $\mathbf{x}^P$ in the direction of the interface, with its magnitude being equal to the value of the level-set function at the particle. Given this definition and recalling that the level-set function is a distance function, $(\mathbf{x}^P + \mathbf{d}^P)$ can be understood to describe a point on the interface, as illustrated in Figure 5. A particle P is then able to give a local first-order (linear) approximation of the level-set function for a position $\mathbf{x}$. This approximation $\phi^P(\mathbf{x})$ is equal to the distance from a point $\mathbf{x}$ to the plane Γ' that passes through $(\mathbf{x}^P + \mathbf{d}^P)$ and is perpendicular to $\mathbf{d}^P$. This suggests that, except for its sign, $\mathbf{d}^P$ represents a vector that is normal to the interface Γ at $(\mathbf{x}^P + \mathbf{d}^P)$.

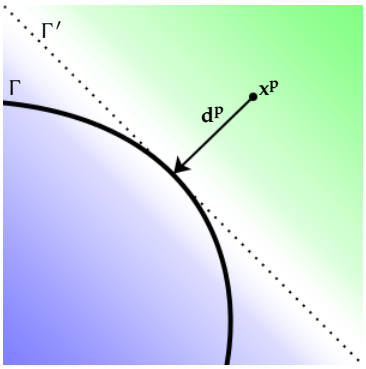


Figure 5: In the OPLS method, a particle P positioned at $\mathbf{x}^P$ and having a direction-vector $\mathbf{d}^P$, provides a linear approximation of the interface denoted as Γ'. A color-gradient visualizes the approximation of the level-set function ϕ^P.

Whenever a particle escapes, i.e its sign differs from that of the level-set function evaluated at the particle's position ($\mathrm{sign}(P) \neq \mathrm{sign}(\phi(\mathbf{x}^P))$), this approximation is used to locally correct ϕ.

Starting from $\phi|_{t_0}$, a set of particles is randomly positioned within a band near the interface. Each particle P is given the sign of $\phi(\mathbf{x}^P)|_{t_0}$, and its orientation vector $\mathbf{d}^P$ is constructed using

$$\mathbf{d}^P = -\phi(\mathbf{x}^P)\frac{\nabla\phi}{\|\nabla\phi\|}(\mathbf{x}^P), \tag{3.4}$$

where $\phi = \phi|_{t_0}$. Recalling Equation 1.7, we can see that $\mathbf{d}^P|_{t_0}$ indeed represents a vector that is normal to the interface.

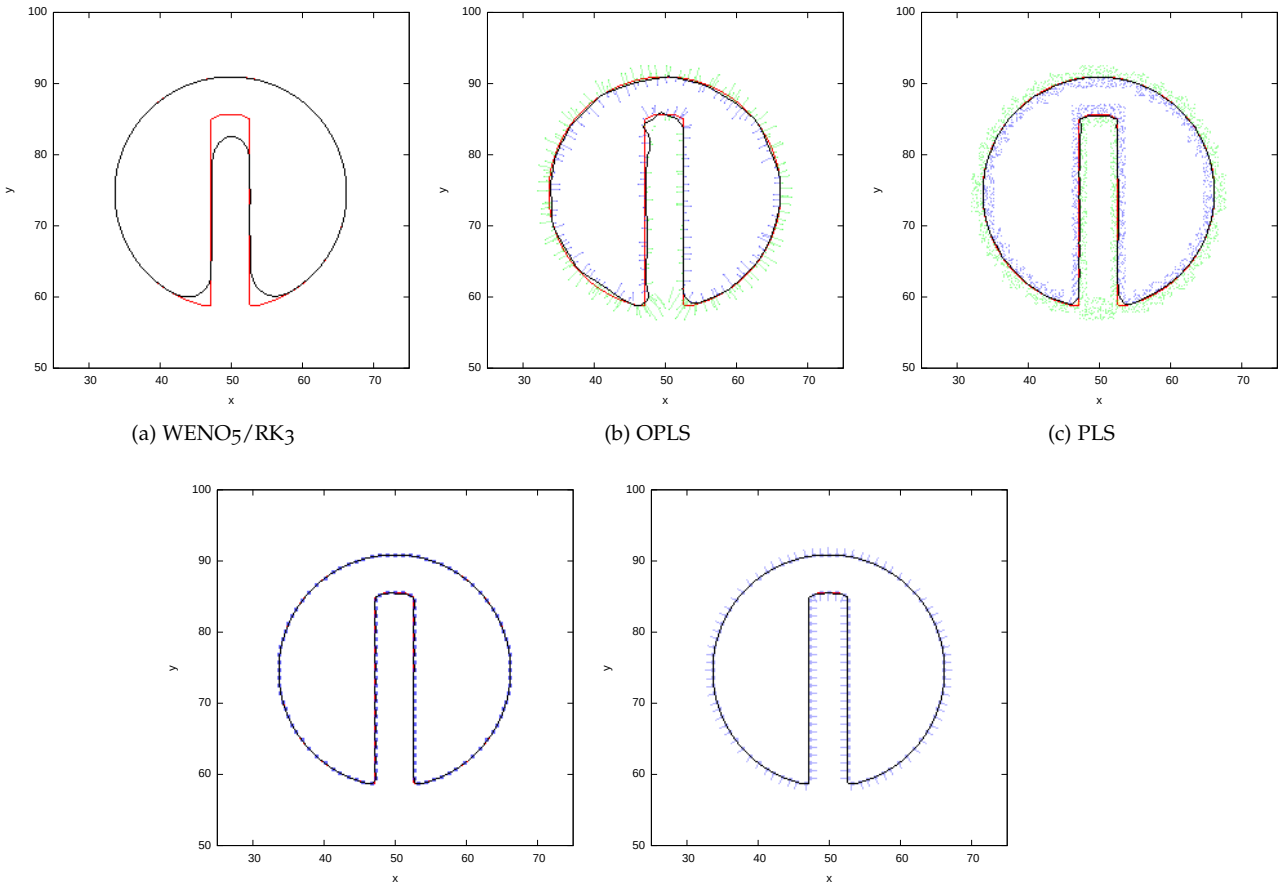
The time-evolution of a particle P is then given by a separate integration of the positions $\mathbf{x}^P$ and $(\mathbf{x}^P + \mathbf{x}^d)$ using Equation 1.1. Due to the rigid-body rotation in Zalesak's Disk test-case this does not lead to problems as $\mathbf{d}^P$ stays normal to the interface. The velocity-field used for the Vortex-in-a-box test however causes $\mathbf{x}^P$ and $(\mathbf{x}^P + \mathbf{d}^P)$ to drift apart and the orientation vector $\mathbf{d}^P$ therefore not to stay normal to the interface. This leads to an incorrect approximation, and therefore correction, of the level-set function by P. To solve this, $\mathbf{d}^P$ is re-set after each correction step using Equation 3.4.

This re-setting step also explains the odd (green-colored) vectors, visible near the tail of the spiral in Figure 7c. These are the orientation vectors belonging to particles that were once close to the tail. Since the level-set function can no longer accurately describe this tail, they have been re-set to the points on the spiral closest to them.

Unfortunately, re-setting $\mathbf{d}^P$ also leads to a circular dependency between the particles and level-set, i.e. the particles influence (correct) the level-set function and vice versa. This circular dependency is problematic as it can cause errors to propagate and accumulate. We shall return to this issue when discussing particle addition/deletion. Note that, in order to prevent this circular dependency, the radii-adjustment step in the PLS method as described in [9] is not enabled in the work of Mokberi and Faloutsos[23] which provided the basis for the implementation used in these tests.

The remaining three methods (PLS, MLS and SA-OPLS) show good corrective capabilities, reducing area-loss to well below 1% in both tests as Table 1 and Table 2 show. The PLS method however needs

more particles to achieve comparable results w.r.t area preservation. This is to be expected since the PLS method places particles in a band near the interface as opposed to MLS and SA-OPLS, which only place particles on the interface itself. Increasing the number of particles, besides adding to the computational burden, appears to have an undesirable side-effect in PLS. As Table 2 shows, increasing the number of particles from 6320 to 37920 improves area-preservation using the PLS method. Comparing the visual results at maximum stretch in Figure 7g and Figure 7e however shows a deteriorated interface when using more particles. Enabling the aforementioned radii-adjustment step does not improve these results.



(a) WENO5/RK3 (b) OPLS (c) PLS

(d) MLS (e) SA-OPLS

Figure 6: Zalesak's Disk after one revolution (at $t = \tau$). The initial disk is shown in red. In *(b)*, orientation vectors are indicated in green and blue depending on the sign of the particle. In *(c)*, particles are colored depending on their sign. In *(d)*, markers are indicated in blue. In *(e)*, particles and normal vectors are indicated in blue.

3.3 PARTICLE ADDITION/DELETION

As an evolving interface stretches, the distance between particles placed near (or on) this interface increases, i.e., the particle density locally decreases. Sections of the interface in which this particle density is too low are not affected by the particle correction step. Such sections consequently give rise to the problem of losing mass. As an example of this, results for MLS with a low, but initially reasonable number of particles (84), are included in Table 2 and shown in Figure 7i and Figure 7j. These results show that the loss of mass is so profound that when particle density increases again, the particles are not able to repair the damage caused as the interface stretched.

For the Vortex-in-a-box tests described above, this problem has been addressed by simply increasing the amount of initially placed particles. Obviously, a more generic solution is to dynamically add and remove particles when needed.

Method	*Number of Particles*	*Area (in % at $t = \tau$)*	*Correction-parameters*	*Particle placement-parameters*
Upwind/Euler	0	**-100,00**		
HJ-WENO5/RK3	0	**+1,01**		
OPLS	425	**-1,36**	$m(a) = a/3$, $\alpha = 1$	$P_{count} = 1$, $I_{mul} = 1$, $B = 1.0$
PLS	3530	**-0,64**		$P_{count} = 1$, $I_{mul} = 10$ $B = 0.6$
MLS	132	**-0,10**	$\rho = 2.0$, $c = 0.01$	$P_{count} = 1$, $L_{thresh} = 0.7$
SA-OPLS	132	**-0.02**		$P_{count} = 1$, $L_{thresh} = 0.7$

Table 1: Configurations and results for Zalesak's Disk test. See text for definitions of parameters appearing in the last two columns.

We first focus on the addition, or seeding, of particles. There are two important aspects in particle addition; *when* to add particles and *how* to initialize them, e.g., how to assign their position or radius. The PLS and MLS methods check the number of particles in each cell to determine whether or not to add new particles According to the description of PLS given in [8], this might be done periodically or depending on some measure of interface stretching and is a problem-dependent choice. The MLS method however, might add particles at each time-step. There is an important similarity between how the PLS and MLS methods initialize added particles. MLS initializes new particles with a position that is obtained by a linear interpolation of the interface using the level set function. PLS randomly places particles inside a band but initializes them with a value for their radius that is obtained from the level set function. In both cases, these properties that are obtained using the level set function are also the properties used in the particle correction step. As a result, both of these methods introduce the circular dependency mentioned in the previous section, that allows propagation and accumulation of errors introduced by the levelset function.

To illustrate the effect of this circular dependency, we run the Vortex-in-a-box test using the MLS method with particle-addition (but not deletion). At each step, the addition-algorithm uses information from the level-set function to determine if, and where, to place new particles. The level-set function however might have a problem accurately describing the interface, such as near the tail of the spiral in Figure 7k for example. This causes a few particles to be added which are inaccurately placed. Such an inaccurately placed particle will participate in the correction of the level-set function at the next time-step, causing more inaccurately placed particles to be added. The results of this circular dependency are visible in Figure 8.

The SA-OPLS method does not suffer from this problem. Similarly to the way it tracks a local interface normal, without relying on the level set function, it can also track one or more tangent vectors, as well as the second fundamental tensor. This information can then be used to detect when to place new particles and how to initialize them.

A high density of particles, besides needlessly adding to the computational burden, can in some cases cause the correction process to deteriorate. Particles that are added during the evolution of the interface contain information that was obtained through interpolation/approximation, thus making them less accurate. If such a particle is close to a particle that contains more accurate information, its contribution has a negative influence on the overall accuracy of the correction. The situation becomes even worse when a particle containing more accurate information, is removed in favor of a particle containing less accurate information.

Method	*Number of Particles*	*Area (in % (at $t = \tau$)*	*Correction-parameters*	*Particle placement-parameters*
Upwind/Euler	0	**-100,00**		
HJ-WENO5/RK3	0	**-49,39**		
OPLS	2460	**-0,89**	$m(a) = a/3$, $\alpha = 1$	$P_{count} = 5$, $I_{mul} = 1$, $B = 2.0$
PLS	6320	**-0,50**		$P_{count} = 10$, $I_{mul} = 2$, $B = 1.0$
PLS	37920	**-0,04**		$P_{count} = 40$, $I_{mul} = 3$, $B = 1.0$
MLS	84	**-94,74**	$\rho = 2.0$, $c = 0.01$	$P_{count} = 1$, $L_{thresh} = 0.7$
MLS	1120	**-0,03**	$\rho = 2.0$, $c = 0.01$	$P_{count} = 10$, $L_{thresh} = 0.1$
SA-OPLS	1120	**-0.06**		$P_{count} = 10$, $L_{thresh} = 0.1$

Table 2: Configurations and results for the Vortex-in-a-box test. See text for definitions of parameters appearing in the last two columns.

For the deletion of particles, there are again two important aspects; *when* to remove particles, and *which* particles to remove. PLS and MLS consider particle density per cell and thus decide to remove particles when the number of particles in a cell is above a threshold. Similar to particle addition, this might be done periodically for PLS and at every time-step for MLS. In PLS, particles having the largest deviation between their radius and the level-set function evaluated at their position, are removed first. For MLS, we have adapted the deletion-strategy slightly compared to the description in [21] by taking into consideration the age of a particle. This allows us to, following the discussion above, remove particles that are younger and therefore likely to contain less accurate information first. In SA-OPLS, only original particles, i.e. the oldest particles, are assigned a task as de-seeder. A particle y is removed by a de-seeder x when y is found to contain little or no additional information compared to x. This is done by thresholding both on the distance between x and y and the dot-product of their normals (through the parameters d_{min} and α_{min} respectively).

Results of the Vortex-in-a-box test with particle addition and deletion for MLS and SA-OPLS can be found in Table 3 and Figure 9. While the results w.r.t. area preservation have become slightly worse for MLS, SA-OPLS appears to perform slightly better while using fewer particles when compared to the results in Table 2.

3.4 CHOOSING A METHOD.

As explained in Section 3.2, the OPLS method suffers from structural flaws that lead to poor results, visible for example in Figure 7c. This disqualifies the OPLS method as a suitable candidate for our choice for a hybrid method.

The popular PLS method shows good results w.r.t mass preservation, but appears to have more problems correcting the level-set function such that it can accurately resolve the thin tail of the spiral in the Vortex-in-a-box test (Figure 7e and Figure 7g) when compared to SA-OPLS and MLS (Figure 7m and Figure 7k). Moreover, the PLS method needs more particles to obtain comparable results w.r.t

Method	*Number of Particles*			*Area (in % at $t = \tau$)*	*Correction-parameters*	*Particle placement parameters*	*Adaptiveness-parameters*
	$t = 0$	$t = \frac{1}{2}\tau$	$t = \tau$				
MLS	84	639	93	**-0,85**	$\rho = 2.0$, $c = 0.01$	$P_{count} = 1$, $L_{thresh} = 0.7$	
SA-OPLS	84	679	84	**-0.05**		$P_{count} = 1$, $L_{thresh} = 0.7$	$\sigma = 1.05$ $\beta = 0.80$ $d_{min} = 0.75$ $\alpha_{min} = 0.2$

Table 3: Configurations and results for the Vortex-in-a-box test for MLS and SA-OPLS using particle addition/deletion. The number of particles is given at $t = 0$, maximum stretch ($t = \frac{1}{2}\tau$) and at the end of the period($t = \tau$). See text for definitions of parameters appearing in the last three columns.

SA-OPLS and MLS. This is understandable, as particles are placed in a band around the interface whereas SA-OPLS and MLS only place particles on the interface itself. When considering interfaces in a 3-dimensional domain, this difference only increases.

The SA-OPLS method includes the notion of direction in its correction much like OPLS, yet it does not suffer from the same flaws, as it accurately tracks these directions as the interface normals. The results obtained and number of particles used in SA-OPLS are comparable to those of the MLS method.

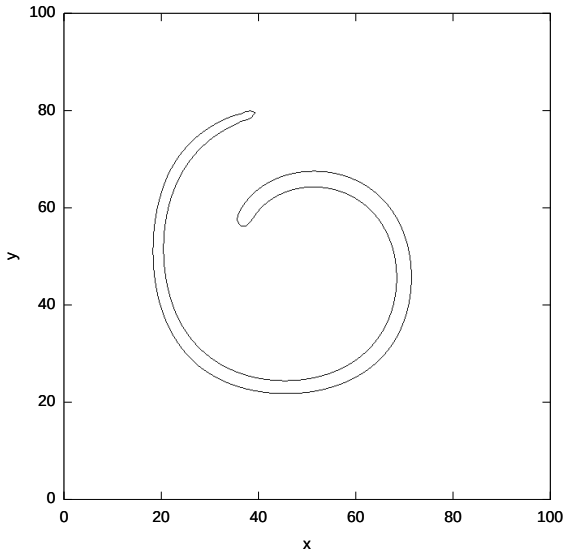
By positioning their particles on the interface, both methods are able to carry surface-information which, as explained in [22], can be useful for example in tracking surface textures. SA-OPLS even tracks more useful information about the local geometry, such as curvature and surface normal, which might be useful for real applications.

The correction step in SA-OPLS is relatively simple, only requiring a dot-product using information from a particle closest to a grid point within a directly neighbouring cell. MLS, on the other hand, is more computationally involved, requiring a weighted average of the contribution of all particles within a potentially wider range. Keeping in mind that we will be parallelizing this step on the GPU, only having to consider a single particle within a fixed limited range appears to be favourable for SA-OPLS.

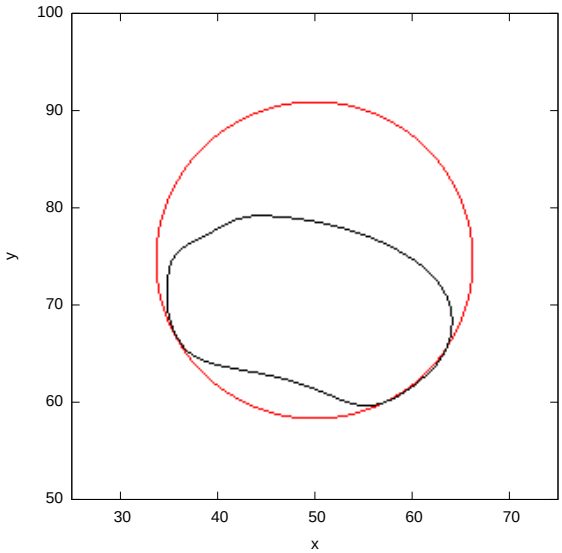
Finally, as explained in Section 3.3, the method of self-adaptiveness proposed for SA-OPLS mitigates a circular dependency between the particles and level-set function, and the potentially harmful side-effects this brings.

Therefore, based on the results presented here and following the discussion above, SA-OPLS is selected as the hybrid method of choice for our implementation.

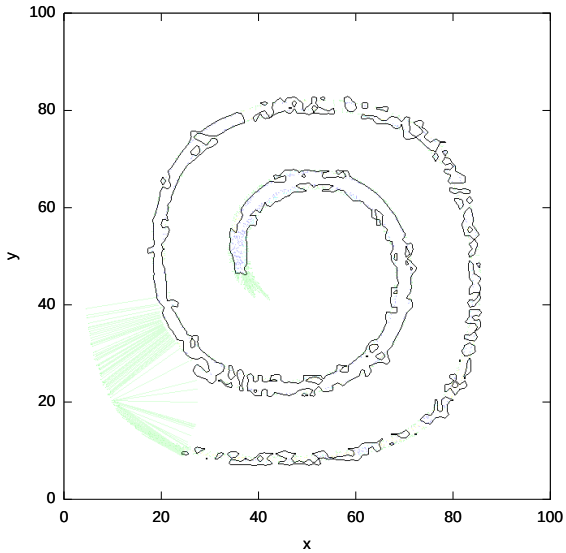
A more in-depth discussion of the SA-OPLS method, its correction and self-adaptiveness, is given in the next chapter.



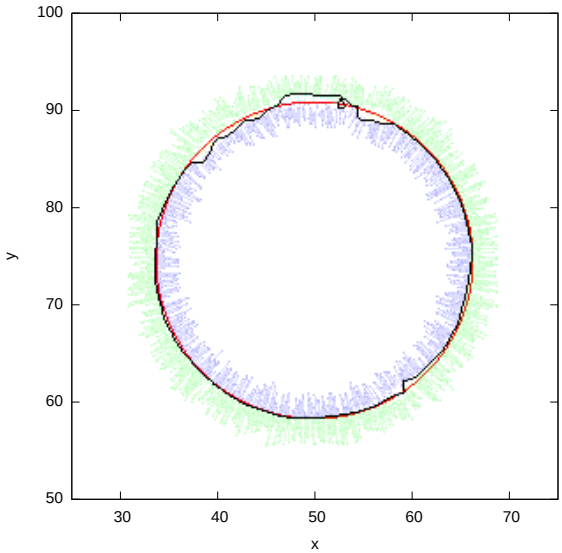
(a) WENO5/RK3 at $t = \frac{1}{2}\tau$



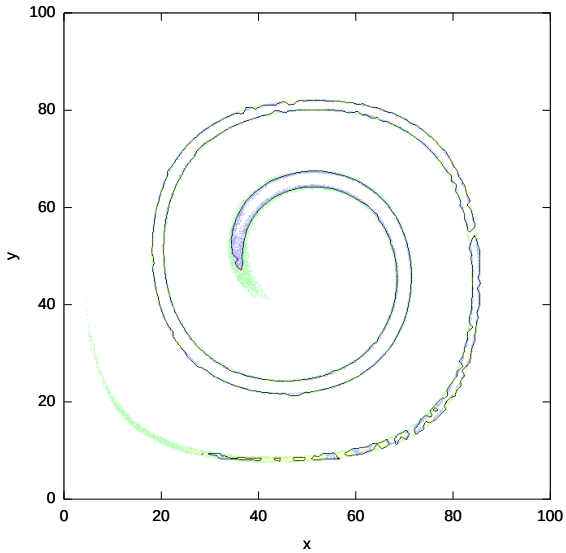
(b) WENO5/RK3 at $t = \tau$



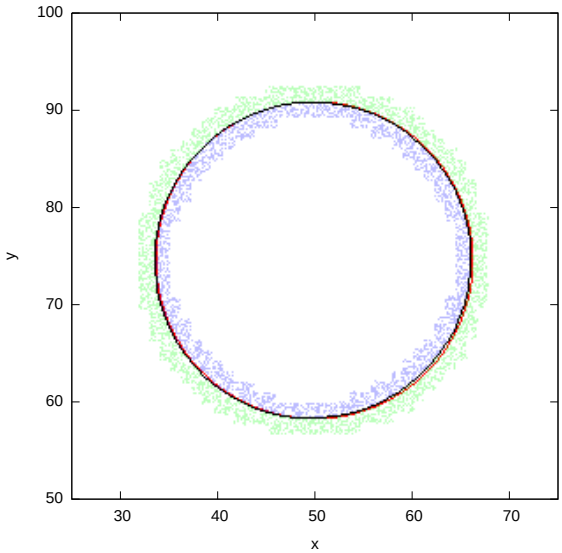
(c) OPLS at $t = \frac{1}{2}\tau$



(d) OPLS at $t = \tau$

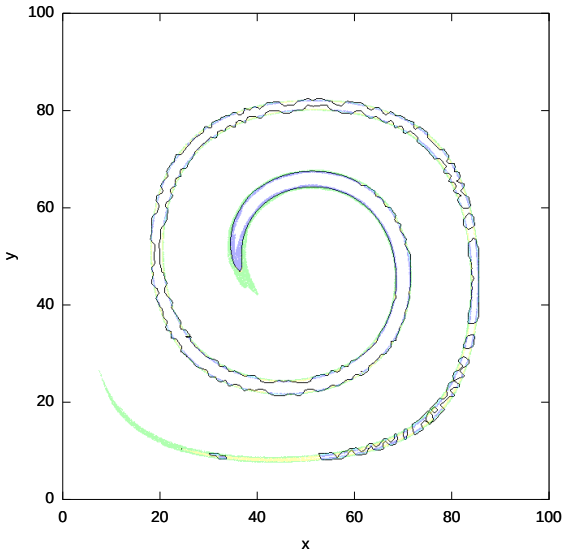


(e) PLS at $t = \frac{1}{2}\tau$ (6320 particles)

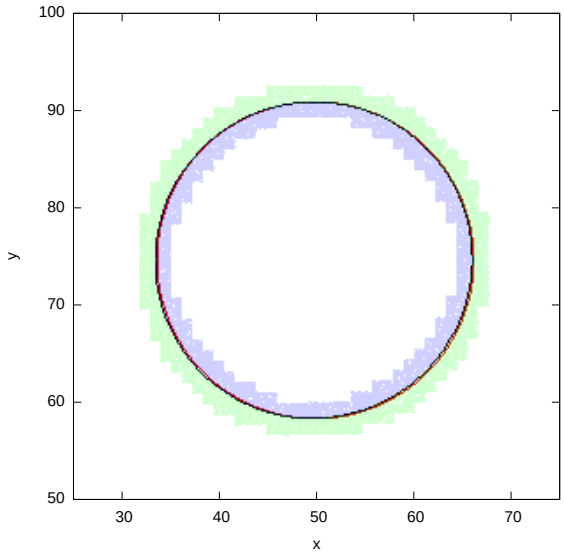


(f) PLS at $t = \tau$ (6320 particles)

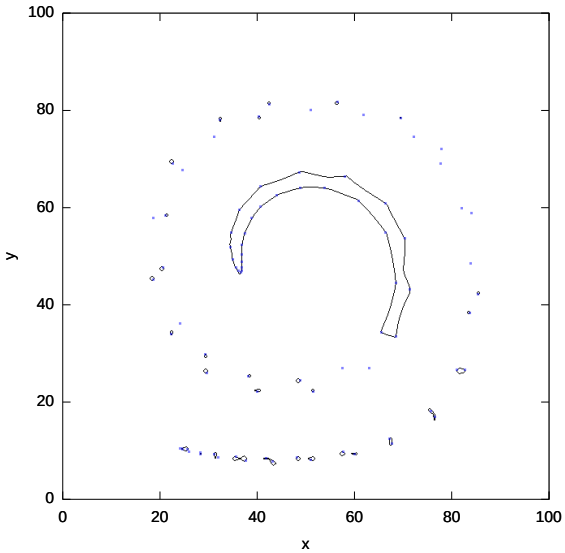
Figure 7: Vortex-in-a-box test at $t = \frac{1}{2}\tau$ (maximum stretch) and $t = \tau$. The original interface at $t = 0$ is shown in red. Direction-vectors in OPLS and particles in PLS are shown in blue and green depending on their sign.



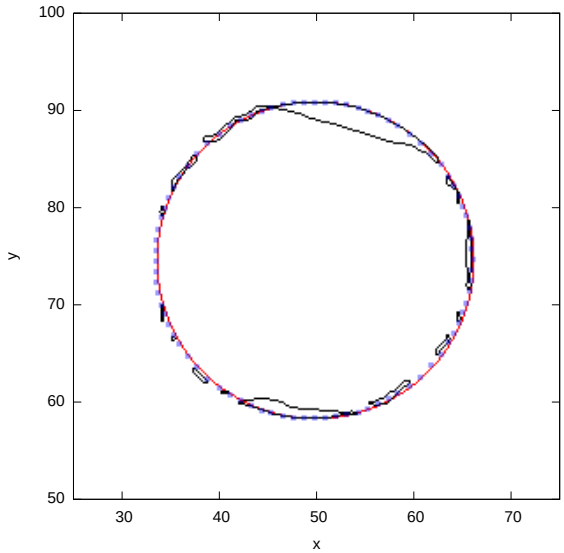
(g) PLS at $t = \frac{1}{2}\tau$ (37920 particles)



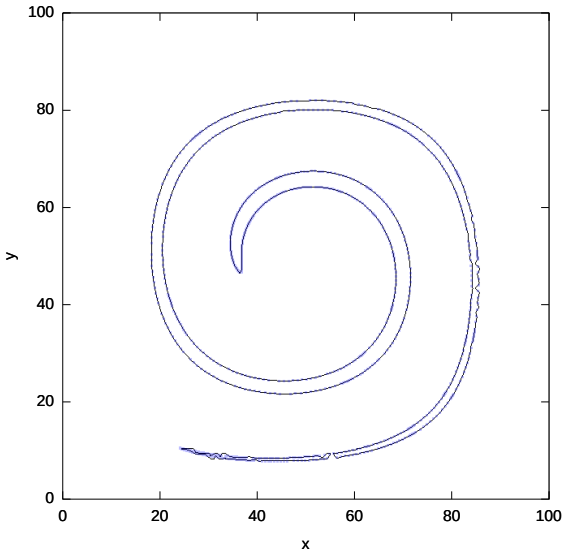
(h) PLS at $t = \tau$ (37920 particles)



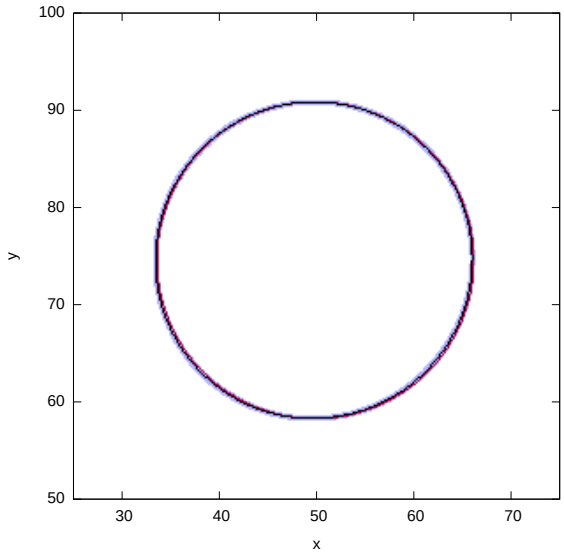
(i) MLS at $t = \frac{1}{2}\tau$ (84 particles)



(j) MLS at $t = \tau$ (84 particles)

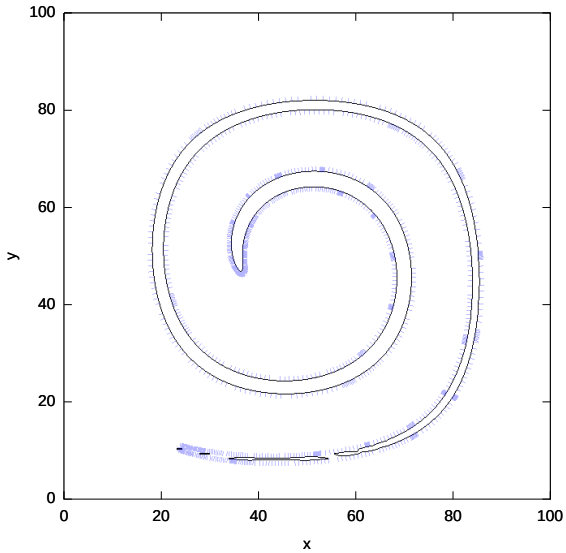


(k) MLS at $t = \frac{1}{2}\tau$ (1120 particles)

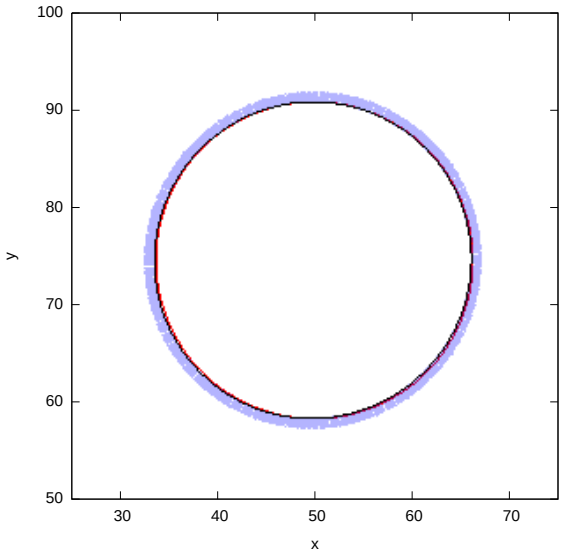


(l) MLS at $t = \tau$ (1120 particles)

Figure 7: Vortex-in-a-box test at $t = \frac{1}{2}\tau$ (maximum stretch) and $t = \tau$. The original interface at $t = 0$ is shown in red. Particles in PLS are shown in blue and green depending on their sign. Particles in MLS, located on the interface, are show in blue.



(m) SA-OPLS at $t = \frac{1}{2}\tau$



(n) SA-OPLS at $t = \tau$

Figure 7: Vortex-in-a-box test at $t = \frac{1}{2}\tau$ (maximum stretch) and $t = \tau$. The original interface at $t = 0$ is shown in red. Particles' normal vectors are shown in blue.

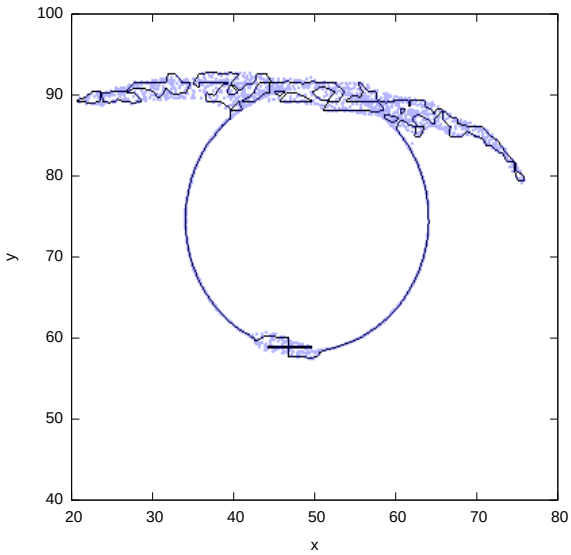
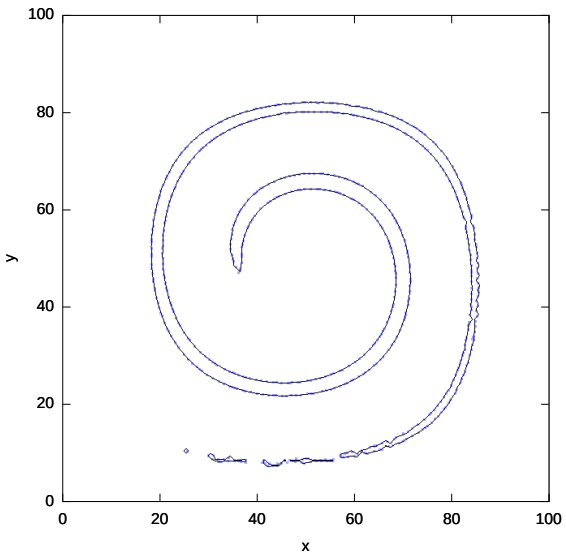
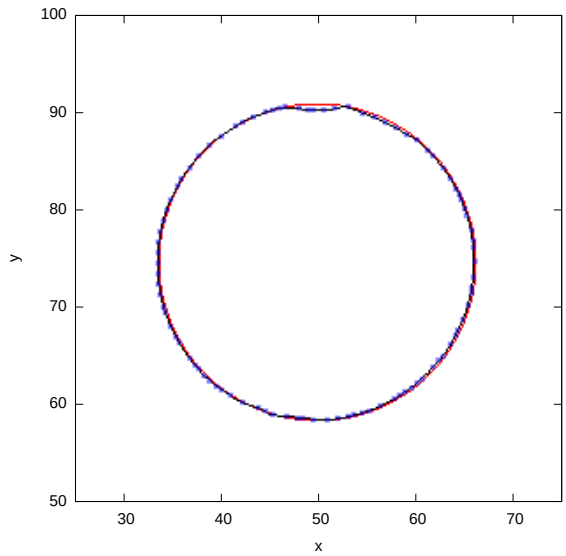


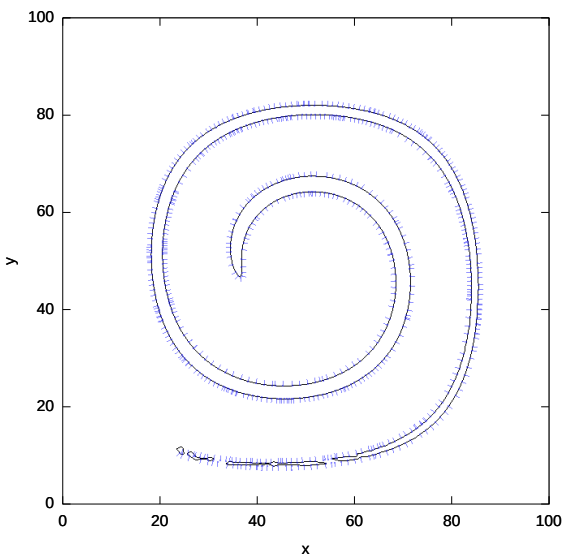
Figure 8: MLS with particle addition (but no deletion) at $t = \tau$ showing the effects of a circular dependency between the particles and level-set function.



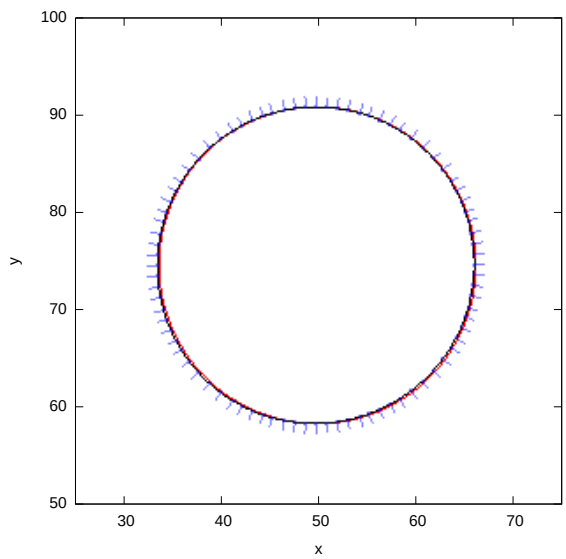
(a) MLS with addition/deletion of particles at $t = \frac{1}{2}\tau$ (639 particles)



(b) MLS with addition/deletion of particles at $t = \tau$ (93 particles)



(c) SA-OPLS with self-adaptiveness enabled at $t = \frac{1}{2}\tau$ (630 particles)



(d) SA-OPLS with self-adaptiveness enabled at $t = \tau$ (84 particles)

Figure 9: Vortex-in-a-box test with particle addition/deletion. The original interface at $t = 0$ is shown in red. Particles, and in SA-OPLS their normal vectors, are shown in blue.

4

THE SELF-ADAPTIVE ORIENTED PARTICLES LEVEL-SET METHOD

This chapter provides an overview of the Self-Adaptive Oriented Particles (SA-OPLS) method as proposed by Ianniello and Mascio [11], and provides the details needed for a discussion of an implementation given in the next Chapter. We start with a description of how oriented particles are used to correct the level-set function ϕ. Section 4.2 then discusses time-evolution of a particle's position and normal. The mechanism for self-adaptiveness is described in more detail in Section 4.3.

4.1 CORRECTION

Ianniello and Mascio initially do not regard the particles as a tool for the correction of the level set function but rather as a method for its construction. They note however that, especially when particle density locally becomes low, a combination of the information carried by the particles with that of a separately advected level-set function improves robustness. We therefore combine the information carried by the particles with that of the level-set function in a correction step similarly to other hybrid methods.

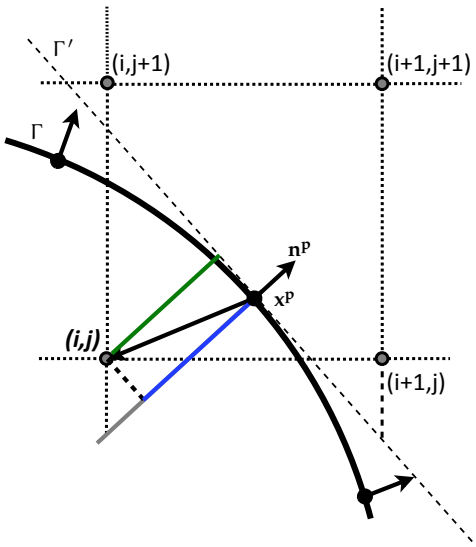


Figure 10: Example of the correction of $\phi(i,j)$ using a first-order, local approximation of the interface denoted by Γ' given by the position of a particle $\mathbf{x}^p$ and its unit normal, $\mathbf{n}^p$. The corrected value ϕ_{corr} (shown in green) is equal to the projection of $((i,j) - \mathbf{x}^p)$ onto $\mathbf{n}^p$ (shown in blue).

An example of how this correction is done is shown in Figure 10. The figure shows one cell of the grid on which the level-set function $\phi(x,y)$ has been discretized. During the correction-step, this grid is updated using local approximations of the interface by the particles. Since these approximations are only accurate locally, a grid point is only updated using information from the particle that is closest to it and that resides in any of the cells of which it is a corner. In the example, $\phi(i,j)$ is the grid point to be updated using particle P. This particle provides a first-order approximation of the interface represented here by a dotted line and denoted by Γ'. Since ϕ was a signed distance function,

the goal is to obtain ϕ_{corr}, the signed distance from (i, j) to Γ'. As the figure shows, this is equal to the projection of $((i, j) - \mathbf{x}^p)$ on $\mathbf{n}^p$, i.e

$$\phi_{corr} = \mathbf{n}^p \cdot ((i, j) - \mathbf{x}^p), \tag{4.1}$$

where $\mathbf{x}^p$ denotes particle P's location and $\mathbf{n}^p$ its unit normal.

4.2 PARTICLE ADVECTION

Given an initial position $\mathbf{x}^p|_{t_0}$, recall from Equation 1.1 that the time evolution of a particle's position is given by

$$\frac{dx^p}{dt} = \mathbf{u}, \tag{4.2}$$

where $\mathbf{u}$ is a velocity-field possibly depending on temporal, spatial or local parameters as mentioned in Section 1.1. This equation can be solved using a second-order Runge-Kutta scheme as given in Equation 3.3.

From the discussion in Section 3.2 it followed that advecting a normal-vector by simply separately advecting its base and tip, as the OPLS method suggests, does not work. However, as Ianniello and Mascio show, an evolution-equation for a particle's unit normal can easily be derived. Let the material derivative of a unit normal vector $\mathbf{n}$ be defined as

$$\frac{D\mathbf{n}}{Dt} = \frac{\partial \mathbf{n}}{\partial t} + \mathbf{u} \cdot \nabla \mathbf{n}. \tag{4.3}$$

Recalling from Equation 1.7 that, since the level-set function is a signed distance function, the outward unit normal can be defined using

$$\mathbf{n} = \frac{\nabla \phi}{\|\nabla \phi\|}. \tag{4.4}$$

Substituting Equation 4.4 in Equation 4.3 gives

$$\frac{D\mathbf{n}}{Dt} = \frac{\partial}{\partial t}\left(\frac{\nabla \phi}{\|\nabla \phi\|}\right) + \mathbf{u} \cdot \nabla \left(\frac{\nabla \phi}{\|\nabla \phi\|}\right), \tag{4.5}$$

which seems to indicate that, in order to solve the evolution equation for a particle's unit-normal $\mathbf{n}^p$, it is necessary to evaluate ϕ. This would lead to a circular dependency between the level-set function and the particles which, as discussed in Chapter 3, is not desirable. Fortunately, by rewriting Equation 4.5 and using the evolution equation for ϕ given in Equation 1.10, it is possible to eliminate ϕ from this equation and obtain an expression for the evolution of $\mathbf{n}$ only in terms of its own components and spatial derivatives of $\mathbf{u}$ (we refer the interested reader to [11] for the derivation). Let a_x denote the x-component of a vector $\mathbf{a}$, and $b_{,y}$ be a shorthand for the spatial partial derivative $\frac{\partial \mathbf{b}}{\partial x_y}$. Then the evolution of $\mathbf{n}$ can be expressed as

$$\frac{D\mathbf{n}}{Dt} = \sum_j \left(\sum_i \left(-u_{j,i} n_j \mathbf{e_i}\right) + \sum_k \left(u_{j,k} n_j n_k \mathbf{n}\right) \right), \tag{4.6}$$

where $\mathbf{e_i}$ is the base unit vector in which $e_i = 1$. Using Einstein's summation convention on repeated indices, Equation 4.6 can be written more concisely as

$$\frac{D\mathbf{n}}{Dt} = -u_{j,i} n_j \mathbf{e_i} + u_{j,k} n_j n_k \mathbf{n}. \tag{4.7}$$

This equation can then be solved in the same way as the evolution equation for a particle's position, i.e., only requiring an initial value and using a second-order Runge-Kutta scheme. For the spatial derivatives of the velocity-field $\mathbf{u}$, first- and second-order central differencing is used, i.e.,

$$\frac{\partial \mathbf{u}}{\partial x_i} = \frac{\mathbf{u}(\mathbf{x}+\mathbf{e_i}\Delta i) - \mathbf{u}(\mathbf{x}-\mathbf{e_i}\Delta i)}{2\Delta i} \tag{4.8}$$

$$\frac{\partial^2 \mathbf{u}}{\partial x_i^2} = \frac{\mathbf{u}(\mathbf{x}+\mathbf{e_i}\Delta i) - 2\mathbf{u}(\mathbf{x}) + \mathbf{u}(\mathbf{x}-\mathbf{e_i}\Delta i)}{\Delta i^2} \tag{4.9}$$

$$\frac{\partial^2 \mathbf{u}}{\partial x_i \partial x_j} = \frac{\mathbf{u}(\mathbf{x}+\mathbf{e_i}\Delta i+\mathbf{e_j}\Delta j) - \mathbf{u}(\mathbf{x}+\mathbf{e_i}\Delta i-\mathbf{e_j}\Delta j) - \mathbf{u}(\mathbf{x}-\mathbf{e_i}\Delta i+\mathbf{e_j}\Delta j) + \mathbf{u}(\mathbf{x}-\mathbf{e_i}\Delta i-\mathbf{e_j}\Delta j)}{4\Delta i \Delta j} \tag{4.10}$$

Please note that, in the tests that were described in Section 3.1 as well the test that will be described in Section 7.2, the velocity-field is given by a continuous function. In case only a discrete velocity-field is given, an intermediate interpolation-step is needed.

4.3 SELF-ADAPTIVENESS

4.3.1 *Particle addition*

As mentioned in Section 3.3, there are two important aspects to particle addition: *when* to add particles and *how* to initialize them, eg., assign their position and any additional properties such as the outward unit-normal.

Intuitively, particles should be added when the interface stretches as this causes the distance between particles to increase and the density, therefore, to decrease. Simply evaluating the distance between a particle and its closest neighbour according to Euclidean distance can give inaccurate results in regions where thin filaments occur, as is the case with the tail of the vortex-in-a-box test. The reason for this is that the closest particle (according to Euclidean distance) is not necessarily a neighbour on the interface. This would imply that a relation among particles should be maintained, but as already mentioned in Section 1.2.1 this becomes very troublesome when topology changes. Alternatively, it is possible to consider the information provided by the normals, since neighbouring particles on the interface are likely to have similar normals.

However, Ianniello and Mascio avoid evaluating inter-particle distances all together and propose a much simpler method for locally estimating stretch. Let each particle be equipped with one or more tangent vectors $\mathbf{t^p}$ (the choice of these tangents will be discussed later on). These tangent vectors are then evolved using an evolution equation that depends on the deformation of the velocity field and, consequently, to the local deformation of the interface. After assigning them an initial value, tangent vectors are evolved using

$$\frac{\partial t_i}{\partial t} = \frac{\partial u_i}{\partial x_j} t_j, \tag{4.11}$$

(again using Einstein's notation) where t_i is one component of a tangent vector $\mathbf{t}$ and $\frac{\partial u_i}{\partial x_j}$ is the spatial partial derivative of u_i with respect to j. To demonstrate how this tangent relates to local stretch, Figure 11 shows the results of the vortex-in-a-box test in which tangents have been visualized. This shows that the magnitude of the tangent vector can be used as an estimate for the amount of local stretch σ^p, i.e.

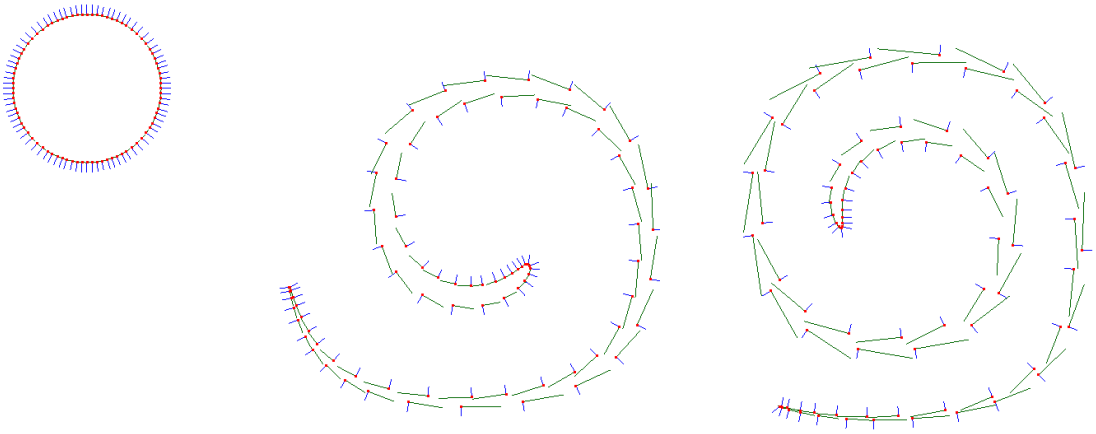
$$\sigma^p = ||\mathbf{t^p}|| \tag{4.12}$$

A new particle is then added after a time-step for each existing particle where

$$\sigma^p \geqslant \sigma_{max}, \tag{4.13}$$

after which the seeding particle resets the magnitude of its tangent.

Having a strategy for deciding when to add a new particle, it is now time to turn to the problem of how to assign a correct position, outward unit-normal and other properties to a new particle. As a new



(a) $t = 0$ (b) $t = 0.25T$ (c) $t = 0.5T$

Figure 11: Vortex-in-a-box test using SA-OPLS showing particles (red), their local outward unit-normals (blue) and tangent (green) vectors at $t = 0$, $t = 0.25T$ and $t = 0.5T$.

particle is to be placed on the interface and will be positioned relatively close to the seeding particle, Ianniello and Mascio propose to base this placement on a second-order, local approximation of the interface, relying only on information provided by the seeding particle.

Let $\mathbf{x^p}$ be the position and $\mathbf{n^p}$ be the normal of a seeding particle P. The goal is to obtain a position $\mathbf{x^q}$ for a new particle Q. First, a Taylor series expansion of the level-set function around $\mathbf{x^p}$, denoted by ϕ^p, up to second-order terms is constructed.

$$\phi^p(\mathbf{x}) = \phi(\mathbf{x^p}) + \nabla\phi|_{\mathbf{x^p}} \cdot (\mathbf{x} - \mathbf{x^p}) + \frac{1}{2}(\mathbf{x} - \mathbf{x^p})^T \nabla(\nabla\phi)|_{\mathbf{x^p}}(\mathbf{x} - \mathbf{x^p}) \tag{4.14}$$

Let the second fundamental form be defined as $\mathbf{T} = \nabla\mathbf{n} = \nabla(\nabla\phi)$. This second fundamental form $\mathbf{T}$, also known as the shape tensor, is a rank-2 tensor which is expressed in terms of components (again using Einstein's notation) as

$$\phi_{,ij} = n_{i,j}\|\nabla\phi\| + n_i n_k \phi_{,kj}, \tag{4.15}$$

where $\phi_{,ij}$ is a short-hand notation for $\frac{\partial^2\phi}{\partial x_i \partial x_j}$ and $n_{i,j} = \frac{\partial n_i}{\partial x_j}$.

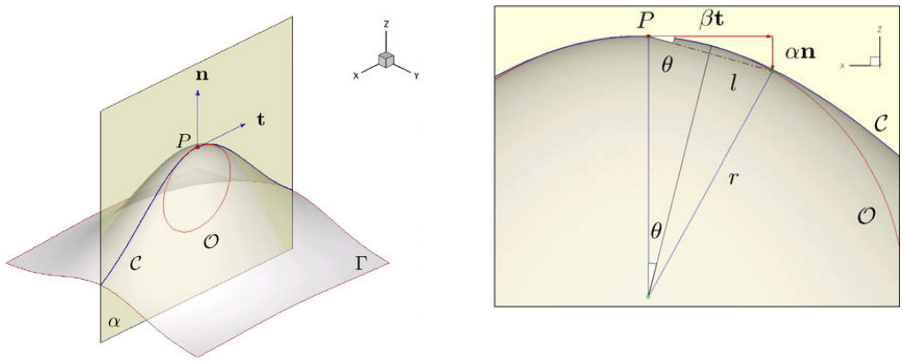
Let $\mathbf{t}$ be a unit vector tangent to the interface Γ at $\mathbf{x^p}$. Together with $\mathbf{n^p}$, $\mathbf{t}$ spans a plane in which a curve $\mathcal{C}$ upon Γ exists as shown in the left of Figure 12. The goal now is to find $\mathbf{x^q}$ on the circle $\mathcal{O}$, osculating $\mathcal{C}$ in this plane at $\mathbf{x^p}$. This allows $(\mathbf{x^q} - \mathbf{x^p})$ to be expressed as

$$\mathbf{x^q} - \mathbf{x^p} = \beta\mathbf{t} + \alpha\mathbf{n^p}, \tag{4.16}$$

where β is a configuration-parameter, and α is used to place $\mathbf{x^q}$ on $\mathcal{C}$. A graphical interpretation of this can be found in the right of Figure 12. It can be seen that increasing β allows control of the distance between the existing and new particle. The larger this distance however, the less accurate the approximation of the interface and therefore placement of the new particle Q is.

An expression is then derived for α. We refer the interested reader to [11] for the full derivation and provide a sketch for it here.

- Substitute $(\mathbf{x^q} - \mathbf{q^p})$ from Equation 4.16 for $(\mathbf{x} - \mathbf{x^p})$ in Equation 4.14,
- use Equation 1.7 for an expression for $\nabla\phi$,
- from geometrical considerations, shown in the right of Figure 12, find the evaluated order of magnitude for α w.r.t β, allowing the intermediate expression for $\phi^p(\mathbf{x})$ to be rewritten up to second order,



(a) A unit vector $\mathbf{t}$, tangent to Γ at $\mathbf{x}^p$, together with $\mathbf{n}^p$ spans a plan containing a curve $\mathcal{C}$ on the interface Γ.

(b) The configuration-parameter β influences the distance between the new and the existing particle P. The unknown α is used to place the new particle on $\mathcal{O}$, the circle osculating $\mathcal{C}$ at $\mathbf{x}^p$.

Figure 12: *from: Ianniello and Mascio [11]*

- now substitute for $\phi_{,ij}$ from Equation 4.15.

Finally, assuming that $\phi(\mathbf{x}) = 0$ for any point $\mathbf{x}$ on the interface, and that $\|\nabla\phi\| \neq 0$, results in

$$\alpha = -\frac{1}{2} n^p_{i,j} t_i t_j, \tag{4.17}$$

where $-n^p_{i,j} t_i t_j$ expresses the curvature projected onto the plane containing $\mathbf{n}^p$ and $\mathbf{t}$, also known as the normal curvature $\tilde{\kappa}$.

$$\tilde{\kappa} = -n^p_{i,j} t_i t_j = -\mathbf{t} \cdot (\nabla \mathbf{n}^p \cdot \mathbf{t}) \tag{4.18}$$

Combining Equation 4.17, Equation 4.18 and Equation 4.16 leads to the following expression for the position of a new particle Q,

$$\mathbf{x}^q = \mathbf{x}^p + \beta \mathbf{t} + \frac{1}{2} \tilde{\kappa} \mathbf{n}^p. \tag{4.19}$$

Taking $\mathbf{t} = \frac{\mathbf{t}^p}{\|\mathbf{t}^p\|}$, where $\mathbf{t}^p$ is a tangent-vector of P that satisfies Equation 4.13 through Equation 4.12, and assuming a suitable value for β is given, it is possible to evaluate Equation 4.19 and find a position for a new particle Q. This would, however, lead to a circular dependency between the level-set function and the particles through the evaluation of the shape tensor $\mathbf{T} = \nabla \mathbf{n} = \nabla(\nabla\phi)$ using the level-set function ϕ. Ianniello and Mascio therefore derive an evolution equation for this tensor similar to that of the normal and tangent vectors described in Section 4.1. Recall Equation 4.7 written in terms of components, i.e,

$$\frac{Dn_i}{Dt} = u_{j,k} n_j n_k n_i - u_{j,i} n_j. \tag{4.20}$$

Taking the spatial derivative of the left-hand side results in

$$\left[\frac{Dn_i}{Dt}\right]_{,s} = \left[\frac{\partial n_i}{\partial t} + u_j \frac{\partial n_i}{\partial x_j}\right]_{,s} = \frac{\partial n_{i,s}}{\partial t} + u_j \frac{\partial n_{i,s}}{\partial x_j} + u_{j,s} n_{i,j} = \frac{Dn_{i,s}}{Dt} + u_{j,s} n_{i,j}, \tag{4.21}$$

and doing the same for the right-hand side yields

$$\begin{aligned}[u_{j,k} n_j n_k n_i - u_{j,i} n_j]_{,s} =& u_{j,ks} n_j n_k n_i + u_{j,k} n_{j,s} n_k n_i + u_{j,k} n_j n_{k,s} n_i \\ &+ u_{j,k} n_j n_k n_{i,s} - u_{j,is} n_j - u_{j,i} n_{j,s}.\end{aligned} \tag{4.22}$$

Then the evolution equation for $\nabla \mathbf{n}$ is given by

$$\begin{aligned}\frac{Dn_{i,s}}{Dt} =& u_{j,ks} n_j n_k n_i + u_{j,k} n_{j,s} n_k n_i + u_{j,k} n_j n_{k,s} n_i + u_{j,k} n_j n_k n_{i,s} \\ & - u_{j,is} n_j - u_{j,i} n_{j,s} - u_{j,s} n_{i,j},\end{aligned} \tag{4.23}$$

which, after being given an initial value, is evolved through time using the same second-order Runge-Kutta method also used for a particle's position, normal and tangent vectors. Tracking this second fundamental tensor also allows the mean curvature at a particle P's position to be evaluated at each step using

$$\kappa = \mathrm{tr}(\nabla \mathbf{n}^{\mathbf{p}}) = n^p_{i,i}, \tag{4.24}$$

where $\mathrm{tr}()$ evaluates the trace. The outward unit-normal $\mathbf{n}^{\mathbf{q}}$ of the new particle positioned at $\mathbf{x}^{\mathbf{q}}$ is assigned using

$$n^q_i = n^p_i + \frac{\partial n^p_i}{\partial x_j}\left(x^q_j - x^p_j\right), \tag{4.25}$$

and for the newly added second fundamental tensor

$$\nabla \mathbf{n}|_{x^q} \approx \nabla \mathbf{n}|_{x^p} \tag{4.26}$$

As for the choice of the tangents $\mathbf{t}^{\mathbf{q}}$, Ianniello and Mascio describe that different strategies can be adopted. A good choice could be to compute the eigenvectors of the second fundamental tensor to obtain the local tangent directions corresponding to the principal curvatures of Γ at $\mathbf{x}^{\mathbf{q}}$. Alternatively, they describe a more simple approach for interfaces in $\mathbb{R}^3$ by selecting two orthogonal tangent vectors, setting their moduli to a suitable value and including their opposite-signed counterparts. This leads to four tangent vectors and therefore up to four particles could be added by an existing particle.

4.3.2 *Particle deletion*

As the interface shrinks and particles gather together, the amount of redundant information represented by the particles increases, i.e., particles that are close to each other and have (largely) similar normals give the same information regarding the interface. A criterion is then needed to decide which of the particles to remove or keep. From the discussion in Section 3.3, we recall that it is generally a bad idea to remove older particles as they contain more reliable information. Ianniello and Mascio therefore propose to let initially-placed particles act as de-seeders for particles that have been added during the evolution of the interface. A particle P, located at $\mathbf{x}^{\mathbf{p}}$ and having a normal $\mathbf{n}^{\mathbf{p}}$, will be selected for removal by a de-seeder D, positioned at $\mathbf{x}^{\mathbf{D}}$ and having a normal $\mathbf{n}^{\mathbf{D}}$, when the following conditions are met

$$|\mathbf{x}^{\mathbf{D}} - \mathbf{x}^{\mathbf{p}}| \leqslant d_{min} \qquad \text{and} \tag{4.27}$$

$$\mathbf{n}^{\mathbf{D}} \cdot \mathbf{n}^{\mathbf{p}} \geqslant \alpha_{min}. \tag{4.28}$$

Before discussing the design and implementation of the combined GPU-STL and SA-OPLS method on a GPU, the next chapter first provides some background information on GPU programming and the STL method.

5

BACKGROUND ON GPUS AND THE STL METHOD

A short introduction on GPUs and NVIDIA's CUDA framework is given in Section 5.1. Together with an overview of the STL method and its GPU-based implementation, given in Section 5.2, this chapter provides the background information needed to discuss our hybrid implementation in Chapter 6.

5.1 THE GPU PLATFORM AND CUDA FRAMEWORK

5.1.1 *GPUs*

Historically, graphics processing units (GPUs) have, as their name indicates, been developed to assist PCs in graphics-related tasks such as computer games. These tasks are generally characterized by computationally intensive, but highly parallel workloads with an emphasis on high throughput. For example, take a 3D scene from a computer game which consists of a number of objects that are defined by vertices (points in 3D space). A GPU is then responsible for transforming these vertices and their properties, by performing matrix multiplications and interpolations, into pixels on the screen at a rate of about 25 frames per second or more, in order for the viewer to experience any motion as fluent. Optimizations for these workloads are reflected in the GPU-architecture, making it highly data-parallel and allowing it to perform floating point instructions at high throughputs, often expressed in GigaFLOPS (billions of floating point operations per second).

In early GPUs, the pipeline in which transformations took place was fixed. In order to allow for more flexibility and control over this pipeline, parts of the pipeline became programmable through so-called shaders. As GPUs had become quite powerful, this generated an interest in using them for other applications as well, such as processing of scientific data. These applications however still had to rely on the graphics APIs to implement their algorithms.

The trend of introducing more flexibility in the processing pipeline of GPUs has continued to a point where they are now fully programmable. Moreover, frameworks such as NVIDIA's CUDA remove the need for the (ab)use of graphics APIs, making GPUs more accessible for general-purpose computing.

NVIDIA's CUDA is a proprietary framework that enables general-purpose computing on NVIDIA hardware. Alternatives, such as OpenCL, allow general-purpose computing on a wider range of hardware (in fact not only limited to GPUs). However, our focus is on the CUDA framework since it is used for the GPU-STL implementation which forms the basis for our implementation.

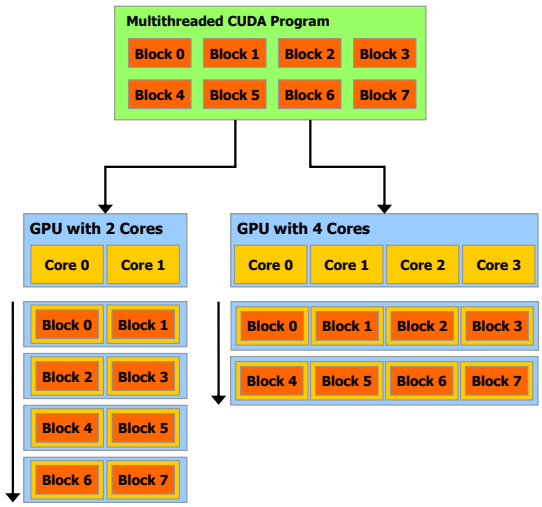
5.1.2 *CUDA*

CUDA exposes the GPU as a fine-grained parallel device, able to run a high number of *threads* in parallel. Each of these threads executes the same function, called a *kernel*, but operates on different data. This results in what is referred to as a Single Instruction Multiple Threads (SIMT) architecture.

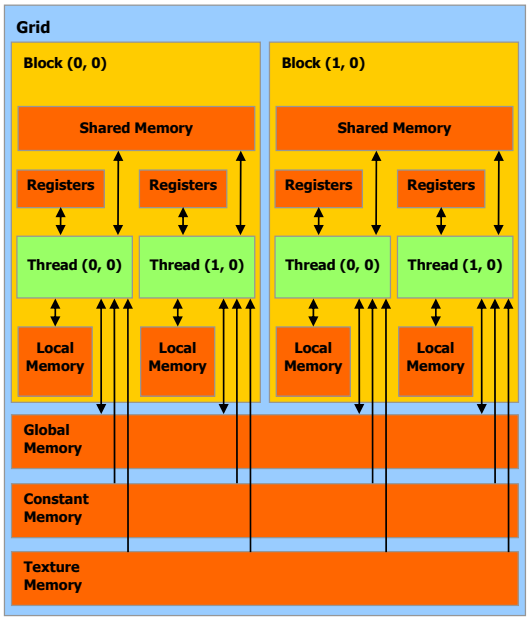
CUDA-enabled GPUs consists of a number of Streaming Multiprocessors (SMs), each of which consists of a number of CUDA-cores. A CUDA-core can be seen as the arithmetic unit that performs the computations for a thread. Obviously, it would be desirable for a CUDA-enabled application to be able to run on any CUDA-enabled GPU (having any number of SMs) while fully utilizing its resources. To allow for this scalability, threads are grouped in *blocks*. The SMs are then responsible for the execution of these blocks as shown in Figure 13a.

A SM is able to service a (limited) number of these blocks in an interleaved fashion with virtually no overhead. This allows a SM to continue processing threads from another block when its current block is

stalled, for example because it is waiting for data from memory. Each SM is equipped with a scheduler that is responsible for selecting the next instruction from one of the resident blocks. Scheduling is in fact not done on block-level, but at a fixed (usually finer) granularity of so-called *warps*.



(a) "A multi-threaded program is partitioned into blocks of threads that execute independently from each other, so that a GPU with more cores will automatically execute the program in less time than a GPU with fewer cores.", *from: CUDA C Programming Guide[1]*. The cores referred to here are the so-called Streaming Multiprocessors (SMs).



(b) An overview of CUDA's memory hierarchy, *from: CUDA C Programming Guide[1]*.

An overview of a CUDA-device's memory hierarchy is given in Figure 13b. The data that a kernel processes usually resides in the global memory, a large RAM that is accessible by all SMs as well as the host-CPU. Constant- and texture memory are read-only optimized global memories. Each SM contains a shared memory that can be used by all threads within a block to cooperate, and a register-file of which it exposes a private part to each thread. Local memory is a part of the global memory that is private to each thread.

The CUDA environment provides an abstraction that makes programming GPUs relatively easy. It is usually less than trivial however, to make efficient use of all the resources provided by the GPU in order to obtain a good performance. Many performance pitfalls are only avoided by a more thorough understanding of the architecture. We briefly discuss here, a number of aspects that impact performance.

- *Occupancy*
  Because a SM can have a number of resident threads, grouped into warps, between which it can quickly switch execution, it is able to hide the latency of warps that stall. It is therefore generally desirable to have as many warps resident as possible, such that the scheduler can always try to find another instruction to execute. The number of warps that a SM can keep in-flight depends on the amount of resources it has available as well as the configuration of the blocks, i.e., how many threads a block consists of. First, the SM simply dictates a fixed upper bound on the number of resident blocks. This would indicate that it is favourable to create blocks with as many threads as possible. Although the number of threads per block also has an upper bound, it is usually not the limiting factor. Because each thread has a state, which consists of the values in the registers it is using, each thread occupies a part of the SM's register file, which is limited in size. One might say that it is therefore generally a good idea to keep the number of registers used per kernel, as that is the function that is executed by each thread, as low as possible. Similarly, if threads in a block cooperate through shared memory, each block occupies a part of the SM's shared memory,

which is again limited in size. To obtain an insight on SM-occupancy and how it is influenced by the capabilities of the device and configuration of the blocks and kernel, NVIDIA has published an *occupancy-calculator*[3].

- *Coalesced memory transactions*
  A SM fetches data from global memory in 32-, 64-, or 128-byte transfers. Assume that a warp consists of 32 threads and that each thread tries to fetch 1 byte of data. If this data is located in memory as a contiguous block of 32 bytes, the SM can fetch it using a single 32-byte transfer. This is called *coalescing*. Suppose the data was not located as a single contiguous block, but scattered over global memory. It would then require 32 separate 32-byte transfers to deliver the data to the threads. Obviously, this greatly reduces memory-bandwidth efficiency as each transfer has included 31 bytes that are not used. There are more conditions that have to be met in order for a fetch to become coalesced, such as alignment, and the order of the addresses with respect to the threads. The reward for meeting these, quite strict, conditions can be significant however. Fortunately, newer CUDA-capable devices (compute-capability 2.x) have less strict conditions for coalescing. Moreover, these devices contain cached global memory, allowing them to exploit spatial locality and in many cases reduce the penalty of non-coalesced accessing.

- *Hiding latency*
  Because global memory is located in off-chip RAM, accessing it takes a considerable amount of time. According to [1], the latency for a memory-transaction is typically between 400 and 800 clock cycles whereas arithmetic instructions take only a few clock cycles. As explained before, this latency can be hidden by the SM by letting it execute instructions from another resident warp. It is therefore important to consider the balance between the number of high-latency memory-related instructions and the low-latency arithmetic instructions. If a kernel only fetches a small amount of data and performs a large number of arithmetic operations on this data, a SM is able to hide memory-latency and the kernel is said to be computation-bound. Conversely, when a large amount of data is transferred but only a small number of arithmetic operations is performed on this data, the kernel is said to be memory-bound. Due to the large difference in latency between memory- and arithmetic instructions, most kernels are memory-bound.

- *Control flow*
  All threads in a warp execute the same instruction in a lock-step fashion. This raises the obvious question of what would happen in case there is any control flow *divergence*, e.g., what if half of the threads in a warp execute the if-part of a conditional and the other half executes the else-part. In that case, all threads within the warp execute all instructions from both parts, i.e., they continue executing in lock-step, but not each thread commits its results. It is therefore important to reduce the amount of control divergence within a warp.

5.2 THE SORTED TILE LIST METHOD

Confidential

6

ALGORITHM DESIGN AND IMPLEMENTATION OF THE HYBRID METHOD USING CUDA

This chapter describes the mapping of the hybrid method on a GPU architecture using CUDA. This implementation is based on GPU-STL[12], described in Section 5.2, which is extended with the SA-OPLS method that was described in Chapter 4.

6.1 PARTICLE ADVECTION

Confidential

6.2 LEVEL-SET CORRECTION

Confidential

6.3 PARTICLE MANAGEMENT

Confidential

6.4 RENDER PARTICLES

Confidential

In the next chapter, results of the *Baseline* implementation described here are presented. A final choice between the cell- and tile-based correction methods is made based on these results. Additionally, a number of improvements are presented and the influence of self-adaptiveness on performance is discussed.

7

RESULTS

In this chapter, the performance of the CUDA-algorithms and implementations described in Chapter 6 are discussed. First, Section 7.1 describes the test-setup on which all tests were performed. Section 7.2 then presents the Enright test, a 3D test that is often found in literature which we used to measure the performance. A description of how particles are initialized for the Enright test is given in Section 7.3. Sections 7.4 through 7.6 then present the results for each of the stages discussed in Sections 6.1 through 6.3. An overview, showing the overhead of self-adaptiveness, is given in Section 7.7. Finally, a comparison with results reported in related work is presented in Section 7.8.

7.1 TEST SETUP

All tests were performed using a low-end consumer graphics card, the NVIDIA GeForce GTS 450. Although this model contains a relatively small amount of CUDA-cores and has a low memory bandwidth, it is based on the new Fermi architecture and has compute-capability 2.1. It can therefore benefit from cached global memory for example. More specifications can be found in Table 4. Version 4.1RC1 of the CUDA toolkit has been used for our implementation.

NVIDIA GeForce GTS 450 *(GF106, Fermi architecture)*	
Compute Capability	2.1
SMs	4
CUDA-cores/SM	48
Clock-speed	1.57 GHz
Memory	1024 MBytes GDDR5
Memory Clock-speed	1.804 GHz
Memory Bus Width	128-bit

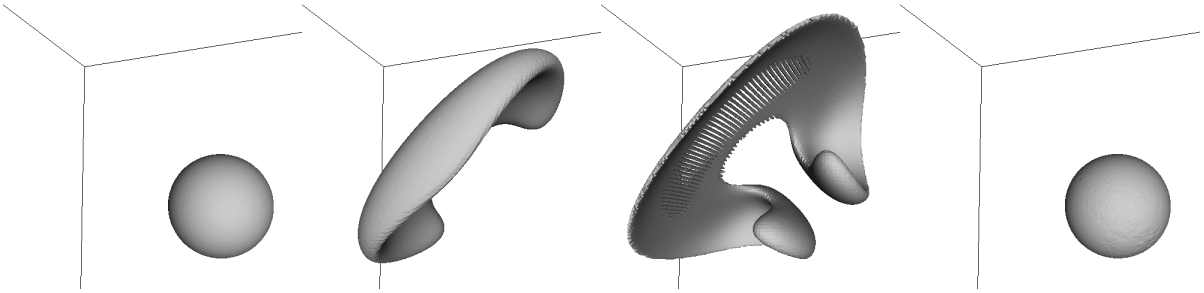
Table 4: Specifications of the CUDA-capable graphics card used for development, testing and benchmarking.

7.2 THE ENRIGHT TEST

To test the performance of our hybrid approach, we use a three-dimensional test that is known as the Enright test. In this test, originally described in[16], a sphere is placed at t_0 in a unit computational domain at $(.35, .35, .35)^T$, having a radius of 0.15. This sphere is then stretched through a periodic incompressible flow field $\mathbf{u}(\mathbf{x}, t)$, defined by

$$\begin{aligned}
u(\mathbf{x}, t) &= 2\sin^2(\pi x_x)\sin(2\pi x_y)\sin(2\pi x_z)\cos(\pi\frac{t}{\tau}),\\
v(\mathbf{x}, t) &= -\sin^2(2\pi x_x)\sin^2(\pi x_y)\sin(2\pi x_z)\cos(\pi\frac{t}{\tau}),\\
w(\mathbf{x}, t) &= -\sin^2(2\pi x_x)\sin(2\pi x_y)\sin^2(\pi x_z)\cos(\pi\frac{t}{\tau}),\\
\mathbf{u}(\mathbf{x}, t) &= (u(\mathbf{x}, t), v(\mathbf{x}, t), w(\mathbf{x}, t))^T,
\end{aligned} \tag{7.1}$$

where $\mathbf{x}$ represents a normalized position in the computational domain, and τ is the period such that the original sphere is recovered at $t = \tau$. Figure 13 provides a few snapshots of the rendered interface as it is deformed in the Enright test. We let $\tau = 3 * G$, where G is the size of the computational domain in a single dimension. The velocity field $\mathbf{u}$ leads to maximum stretch at $t = 0.5\tau$. As the sphere is stretched, parts of the interface approach each other. This makes it difficult for the grid to resolve the interface, much like in the Vortex-in-a-box test. As in the two-dimensional tests discussed in Chapter 3, a first-order upwind scheme is used for levelset advection. Because an additional rescaling speed term was found to be insufficient for maintaining ϕ as a signed distance function, an additional two-iteration reinitialization step is performed after levelset advection using Equation 1.16. Particle advection is done using a second-order TVD Runge-Kutta scheme, given in Equation 3.3, which was also used in the two-dimensional tests. The computational domain is set to 128^3 and a time-step of $\Delta t = 0.1$ time-units is used in all tests.



(a) $t = 0$ (b) $t \approx \frac{1}{4}\tau$ (c) $t \approx \frac{1}{2}\tau$ (d) $t = \tau$

Figure 13: Snapshots of the Enright test at different instants in time.

7.3 PARTICLE INITIALIZATION

To obtain an even distribution of particles over the surface of the initial sphere in the Enright test, the faces of an icosahedron are first recursively sub-divided. When a certain recursion-depth is reached, all resulting vertices are projected onto the surface of the sphere. The number of particles N_{p1} that is positioned in this way is given by

$$N_{p1} = 10 \cdot 2^{2l} + 2, \tag{7.2}$$

where l is the depth of recursion. These particles are supplemented by N_{p2} randomly placed particles, allowing an arbitrary number of $N_p = N_{p1} + N_{p2}$ particles to be used. Having a position $\mathbf{x^p}$, each particle is then assigned its additional quantities as described in [11], starting with a normal $\mathbf{n^p}$ using

$$\mathbf{n^p} = \frac{\mathbf{x^p} - \mathbf{x^C}}{r},$$

where $\mathbf{x^C}$ is the center of the sphere, and r its radius. A first tangent $\mathbf{t_1^p}$ is obtained from the vector product between $\mathbf{n^p}$ and the unit-vector $\mathbf{e_i}$ where $i = min\{|n_j|\}$. The second tangent $\mathbf{t_2^p}$ is then obtained using

$$\mathbf{t_2^p} = \mathbf{n^p} \times \mathbf{t_1^p}.$$

The third and fourth tangents are then

$$\mathbf{t_3^p} = -\mathbf{t_1^p},$$
$$\mathbf{t_4^p} = -\mathbf{t_2^p}$$

The shape tensor $\nabla\mathbf{n}$ is initialized using

$$n_{i,j} = \frac{1}{r}(\delta_{ij} - n_i n_j),$$

where δ_{ij} represents the Kronecker delta.

7.4 PARTICLE ADVECTION

A distinction was made in Chapter 6 between attributes used for correction and those used for self-adaptiveness (SA). In our evaluation of the implementation of the particle advection kernel, we maintain this distinction as it provides an insight into the overhead needed for self-adaptiveness. We first introduce the roofline model[36] with which we are able to evaluate performance and identify possible bottlenecks.

As described in Section 5.1.2, the two most important resources on a GPU that impose a bound on throughput are memory bandwidth and the number of arithmetic units. A roofline model[36] can be used to visualize these bounds for a kernel executing on a GPU. It is important to note that the roofline model is not meant to be an exact or complete model, it merely provides a graphical overview of theoretical limits, thereby providing a frame of reference. To build this model, we first compute the theoretical peak memory bandwidth and single precision floating point throughput for our GTS 450.

The theoretical bandwidth limit BW_{peak} is given by

$$BW_{peak} = F_{clk} * 2 * \frac{B_w}{8}, \tag{7.3}$$

where F_{clk} is the memory clock-speed in GHz and B_w is the Memory bus width in bits. Using the specifications in Table 4 results in

$$BW_{peak} = 1.804 * 2 * \frac{128}{2} = 57.728\,\text{GB/sec}. \tag{7.4}$$

Similarly, the theoretical peak floating-point arithmetic throughput, T_{peak}, is given by

$$T_{peak} = F_{clk} * N_{sm} * (T_{fp} + T_{sfu}), \tag{7.5}$$

where F_{clk} is the GPU's clock-speed in GHz, N_{sm} is the number of streaming multiprocessors (SMs), T_{fp} is the single precision floating-point throughput per SM, and T_{sfu} the throughput of the Special Function Units (SFUs) per SM. These throughputs can be found in Table 5-1 of the programming guide[1]. Note that the single precision floating-point throughput per SM is sometimes reported as $2T_{fp}$ since each CUDA-core can perform a Fused Multiply Add (FMA), consisting of two operations, in one cycle. We assume however that the compiler optimizes floating point arithmetic such that it produces an FMA instruction whenever possible. This allows us to simply count the number of floating-point instructions produced by the compiler without distinguishing between FMA and non-FMA instructions. For the GTS 450 we compute a maximum throughput of

$$T_{peak} = 1.57 * 4 * (48 + 8) = 351.68\,\text{GFLOP/sec}. \tag{7.6}$$

The resulting roofline model, presented in Figure 14, expresses what was already mentioned in Section 5.1.2: if a kernel has low computational intensity (less than approximately 6 FLOP/byte in our case), it is likely to be memory-bound. Conversely, a kernel that has high computational intensity is likely to be computation-bound. While BW_{peak} and T_{peak} impose an upper bound on the attainable throughput, certain characteristics of a kernel's workload can lower these ceilings. Assume for example that a kernel only executes floating-point instructions that are handled by the SFU such as sin(), cos() and inverse square root. This significantly reduces T_{peak}, adding a new ceiling as shown in Figure 14. The ratio between SFU and non-SFU floating-point operations, clearly a characteristic of the implementation, can therefore impose a bound on the attainable throughput. Since CUDA-cores also execute non-floating-point instructions, the ratio between floating-point and non-floating-point operations can impose a similar bound. Certain characteristics of a kernel can also reduce BW_{peak}. Assume a kernel in which each of the 32 threads in a warp fetches 4 bytes of data that are scattered in global memory. On our Fermi-based GPU, this will result in 32 separate 128-byte loads from global memory if each thread suffers a cache-miss. Assuming enough data and an extremely unfavourable access-pattern, all threads might suffer a cache-miss, which effectively reduces memory throughput by a factor of 32 to only 3.125% of BW_{peak}.

Some useful characteristics of our particle advection kernel are gathered by inspection of the disassembled code using *cuobjdump*[2]. As mentioned, we first consider a version of the kernel in which additional quantities needed for self-adaptiveness are not tracked.

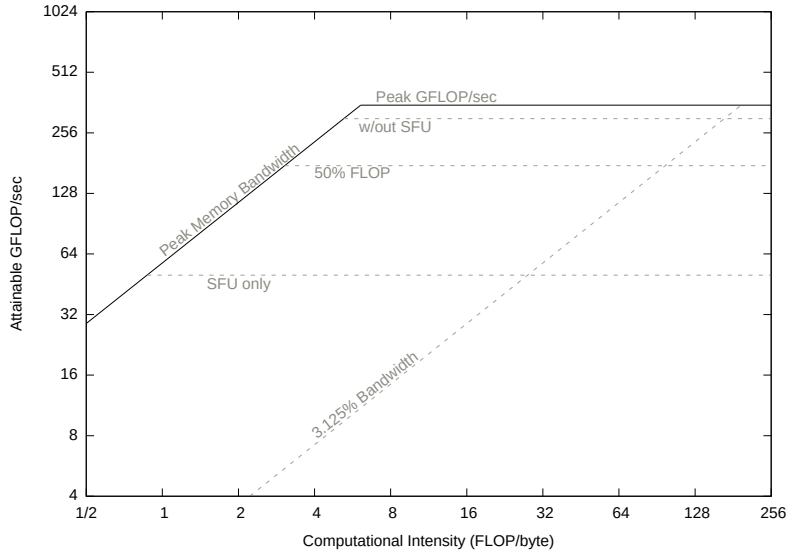


Figure 14: Roofline model for GTS 450. Additional ceilings indicate maximum attainable throughputs given certain characteristics of a kernel. The *w/out SFU*, *SFU only* and 50% *FLOP* ceilings result from computational characteristics of a kernel. The 3.125% *Bandwidth* ceiling results from a unfavourable memory access-pattern in which each 4-byte load would suffer a cache-miss.

7.4.1 *Without Self-adaptiveness*

Recall from Chapter 6 that, in order to perform a correction of the level-set function, a particle contains two quantities which describe its position and normal-vector. Their combined size is 6 *floats*, or 24 bytes. Using *cuobjdump*, we counted the number of *floating-point-*, *memory-* and *other* instructions, the results of which are presented in Table 5.

Particle Advection Kernel without Self-Adaptiveness

Registers	*Floating-point instructions (normal/SFU)*	*Memory instructions (global/local)*	*Bytes transferred*	*FLOP/byte*	*Other instructions*
34	323/100	14/0	56	7.55	40

Table 5: Statistics for the particle advection kernel without SA, gathered from compiler output and *cuobjdump*.

Using the number of registers reported by the compiler, the *occupancy-calculator*[3] suggested us to use 192 threads per block, resulting in 30 active warps per SM, or 63% occupancy.

Each memory instruction reported in Table 5 transfers a single 4-byte word, resulting in the transfer of 56 bytes for each thread. This includes fetching an index from the *active particles* list (4 bytes), loading and storing the particle's data ($2 * 24$ bytes), and writing its tile- or cell-coordinate to the *particle coordinates* list (4 bytes). With a total of 423 floating-point instructions, this results in a computational intensity of 7.55 FLOP/byte.

With these results, we are able to construct the roofline model shown in Figure 15. First, a vertical dashed line is drawn for our kernel at $x = 7.55$. Because this line intersects with T_{peak}, the model indicates that, assuming an optimal mix of instructions, our kernel would be compute-bound. Unfortunately, from Table 5, we know that there are a large number of instructions that need to be executed by the SFU. Where the optimal ratio of non-SFU:SFU instructions is $48 : 8 = 6 : 1$, our kernel had a ratio of $323 : 100 = 3.23 : 1$. Because this ratio is lower than the optimum, we are bound by the throughput of the SFUs. This indicates that the CUDA-cores operate at $\frac{3.23}{6} \cdot 100 \approx 54\%$ of their maximum throughput. An additional SFU-bound ceiling of

$$T_{SFU-bound} = F_{clk} * N_{sm} * \left(\left(\frac{I_{non-SFU}}{I_{SFU}} + 1 \right) * T_{sfu} \right) = 1.57 * 4 * ((3.23 + 1) * 8) = 236.88 \text{GFLOP/sec},$$

(7.7)

where I_{SFU} and $I_{non-SFU}$ are the number of SFU- and non-SFU instructions, is therefore added to the roofline model in Figure 15.

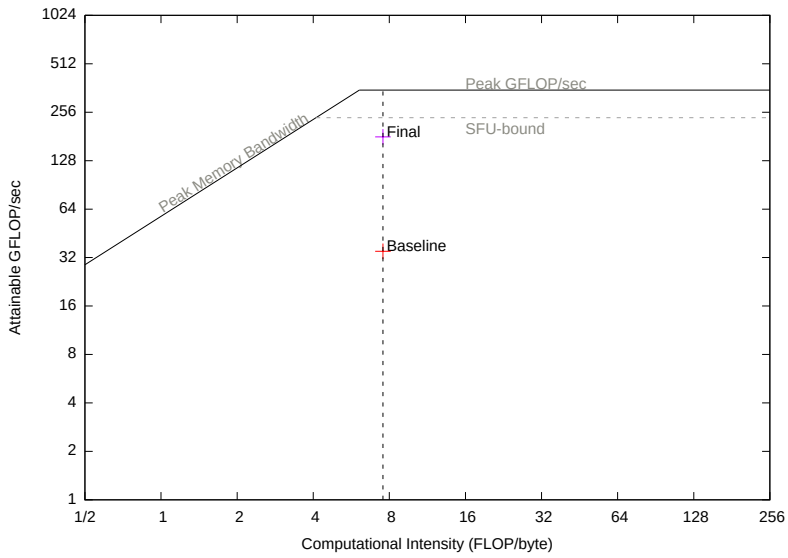


Figure 15: Roofline model of the Particle Advection kernel without self-adaptiveness on a GTS450. An additional SFU-bound, given in Equation 7.7, is shown. Additionally, the performances of the *Baseline* and *Final* implementations are shown.

Note that the high number of instructions to be executed by the SFUs originate from the sin() and cos() functions in the equations for the velocity field in the Enright test given in Equation 7.1. In many real-world applications, this velocity-field will not be defined as an analytical function, but obtained through interpolation on a velocity-grid.

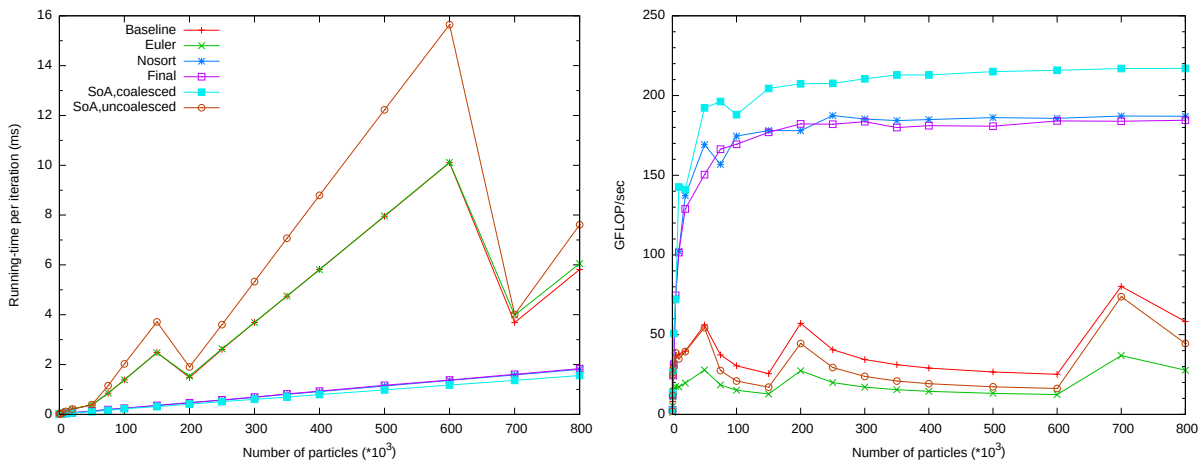
Because the CUDA-cores are under-utilized, the additional $\frac{40}{323} \cdot 100 \approx 13\%$ required by other non-floating-point instructions does not introduce any additional lower ceiling.

We then evaluate the effective throughput of the particle advection kernel by measuring its running-time. The average running-time over 1000 iterations is measured for a varying number of particles. The results of these measurements are presented as the *Baseline* in Figure 16a. As the number of particles remains constant and the number of FLOPs per particle is known, we can compute the effective throughput T_{eff} in GFLOP/sec as follows

$$T_{eff} = \frac{N_p * N_{FLOP/particle} * 10^{-9}}{t * 10^{-3}}, \tag{7.8}$$

where N_p is the number of particles and t is the running-time per iteration in ms as reported in Figure 16a. Results of the effective throughputs are presented in Figure 16b.

By plotting the average achieved throughput of approximately 35 GFLOP/sec for our *Baseline* implementation in the roofline model in Figure 15, it becomes obvious that the kernel does not reach the predicted SFU-bounded ceiling. As no predicted ceiling is reached, it is unclear whether the kernel is computation- or memory-bound. We therefore conduct an experiment in which the amount of transferred data remains unchanged, while the number of floating-point instructions is significantly reduced. If the results show a significant increase in throughput, then the kernel is most likely compute-bound. Conversely, if the results show a significant decrease in throughput, then most likely the kernel is memory-bound. By using a first-order Euler time-step instead of second-order TVD Runge-Kutta, the number of floating-point instructions can be halved, while the amount of data transferred remains unchanged. This was verified using *cuobjdump*. As shown in Figure 16a(*Euler*), this reduction in the number of floating-point instructions does not have a significant influence on running-time. Because $N_{FLOP/particle}$ has been reduced, throughput as reported in Figure 16b, had also significantly reduced. We therefore conclude that the *Baseline* implementation is likely not to be computation-bound, but memory-bound.



(a) Running-time. Note that results for *Baseline* and *Euler* overlap. Similarly, results for *Nosort* and *Final* overlap.

(b) Effective throughput. Computed using Equation 7.8 and the results in *(a)*.

Figure 16: Running-time and corresponding effective throughputs of the Particle Advection kernel without self-adaptiveness for a varying number of particles.

The *Baseline* implementation shows peculiar behavior when the number of particles increases from 150 to 200 thousand and from 600 to 700 thousand. Whereas an increase in the number of particles would logically lead to an increase in running-time, two significant drops can be observed in the aforementioned intervals. To understand this behavior, recall that particles are positioned at t_0 using a combination of structurally and randomly generated positions. According to Equation 7.2, using $l = 7$ we are able to structurally place 163842 particles. In order to place 163841, i.e. one less particle, the structural approach, using $l = 6$, generates 40962 particle-positions. The remaining $163841 - 40962 = 122879$ particles are positioned randomly. Similarly, using $l = 8$ allows structural placement of 655362 particles. From Figure 16a we see that these numbers are strongly related to the peculiar drops in running-time. Now recall that particle-data in our *Baseline* implementation are accessed indirectly through the *active particles* list, and that this list is sorted lexicographically according to cell- or tile-position. Obtaining particle-data by indirection through a sorted list causes scattered memory-accesses and can significantly reduce spatial locality. This effect is amplified when many particles are positioned randomly. With low spatial locality, we are unable to benefit from the L2 cache. To verify this, we perform an experiment in which we skip sorting. The results in Figure 16(*Nosort*), show an impressive reduction of running-time up to almost one order of magnitude.

Obviously, sorting is needed for the correction stage and cannot simply be discarded. Fortunately, there is an easy way to achieve better spatial locality. As the scattered access pattern was caused by sorting of the *active particles* list, we simply bypass indirection through this *active particles* and access particles directly from the *particles* array. The results for this *Final* implementation, presented in Figure 16, verify that the performance is similar to the *Nosort* version. The maximum throughput of approximately 180 GFLOP/sec reported in Figure 16b, is now much closer to the predicted 236 GFLOP/sec SFU-bound ceiling.

Recall from Section 6.1 that an array of structures (AoS) approach was used for storing the particle data. On pre-Fermi architectures, which were not equipped with cached global memory, this was considered bad practice as it often prevented coalesced memory accessing and therefore reduced effective memory bandwidth. A structure of arrays (SoA) approach often allows memory coalescing and was therefore preferred. Above we have shown how we benefit from caching by improving spatial locality. Similar to coalescing, this improves effective memory bandwidth. Since the particle advection kernel can easily be modified to adopt a SoA approach and benefit from coalesced memory accessing, it is interesting to compare these approaches. As shown in Figure 16, the SoA approach slightly improves running-time and achieves a throughput of approximately 210 GFLOP/sec, an improvement of roughly 17%. When coalesced memory accesses are not possible, the SoA approach exhibits less spatial locality and therefore benefits less from caching when compared to the AoS approach. This

results in an average throughput of 25 GFLOP/sec, which is a degradation of roughly 30%. Because it is expected to be more difficult to achieve coalesced access patterns in the correction- and particle management stages, we maintain the AoS approach and rely on caching.

7.4.2 *With Self-adaptiveness*

Similar to our analysis of the particle advection kernel without self-adaptiveness, we begin again with the construction of a roofline model from statistics generated by the compiler and gathered using *cuobjdump*.

First, we briefly discuss the evolution equation for the shape tensor $\mathbf{T}$ given in Equation 7.9. Without the use of Einstein's shorthand notation for summation on repeated indices, this equation is written as

$$\frac{Dn_{i,s}}{Dt} = \sum_j \left(\sum_k \left(u_{j,ks} n_j n_k n_i + u_{j,k} n_{j,s} n_k n_i + u_{j,k} n_j n_{k,s} n_i + u_{j,k} n_j n_k n_{i,s} \right) - u_{j,is} n_j - u_{j,i} n_{j,s} - u_{j,s} n_{i,j} \right). \quad (7.9)$$

Our implementation of this formula contains a loop for each of the indices i, s, j, k, leading to as much as four levels of nested loops. To obtain a fair count of the number of floating-point instructions, these loops are unrolled by the compiler through the use of `#pragma unroll`. Additionally, unrolling these loops might help the compiler in its optimizations.

Comparing the statistics in Table 6 with those reported in Table 5, it can be seen that both the number of operations and amount of data transferred have increased significantly because each particle now additionally tracks four tangent vectors and the shape tensor. Note that the computational intensity remains close to that of the kernel without self-adaptiveness because, while the evolution equation for the shape tensor adds many computations, the evolution of the tangents, found in Equation 4.11, requires relatively few computations.

Particle Advection Kernel with Self-Adaptiveness

Registers	*Floating-point instructions (normal/SFU)*	*Memory instructions (global/local)*	*Bytes transferred*	*FLOP/byte*	*Other instructions*
63	1881/100	58/273	232	8.54	50

Table 6: Statistics for the particle advection kernel with SA, gathered from compiler output and *cuobjdump*.

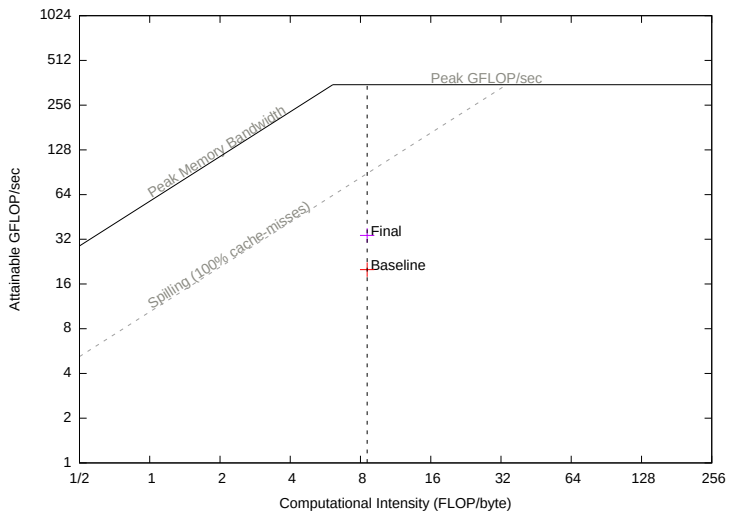
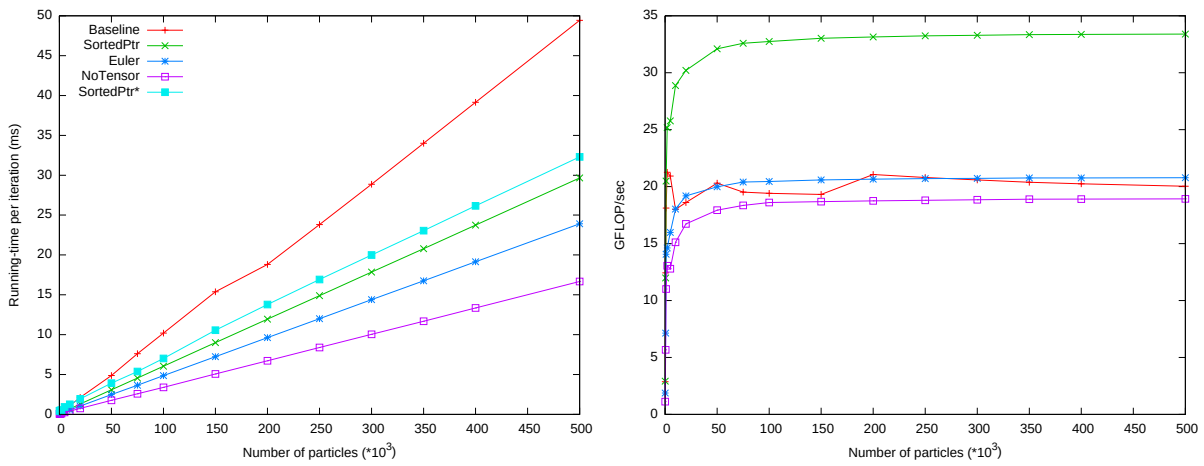


Figure 17: Roofline model of the Particle Advection kernel with self-adaptiveness on a GTS450.

Unlike the kernel that did not track the additional quantities needed for self-adaptiveness, the statistics in Table 6 show that, in this case, relatively few floating-point instructions need to be handled by the SFUs. The kernel is therefore not bound by the throughput of these SFUs.



(a) Running-time.

(b) Effective throughput. Computed using Equation 7.8 and the results in *(a)*.

Figure 18: Running-time and corresponding effective throughputs of the Particle Advection kernel with self-adaptiveness for a varying number of particles.

The number of registers used in this version of the kernel has increased significantly, now requiring 63 registers. This is in fact the maximum number of registers per thread for the Fermi architecture[20]. From the output generated by the compiler, using the `-ptxas-options="-v"` switch, we learn that 63 registers is not even sufficient as register spilling occurs. This can also be seen in the number of memory-related instructions. Although only 58 fetches are needed for loading and storing the particle data, an additional 273 load- and store-instructions are issued. From inspection of the disassembled code, we learn that these additional memory-instructions all operate on local memory, which is where registers are spilled to. Recall that local memory resides in the same off-chip memory as global memory. If all loads and stores from local memory are cache-misses, this can seriously reduce the effective bandwidth. To visualize this, we add the following ceiling to the roofline model in Figure 17,

$$BW_{spill} = \frac{M_{global}}{M_{global} + M_{spill}} BW_{peak} = \frac{58}{58 + 273} BW_{peak} \approx 0.18\, BW_{peak}, \tag{7.10}$$

where M_{global} and M_{spill} are the number of memory-related instructions used for accessing global memory and register spilling respectively.

Because the kernel uses many registers, it will become more difficult for the SMs to achieve a good occupancy. Using the *occupancy-calculator* we find that, at 256 threads per block only 16 warps can be kept active, which leads to an occupancy of 33%.

Running-times and throughput for the *Baseline* implementation of this kernel are found in Figure 18. Note that, while additional quantities needed for self-adaptiveness are tracked, no particles are actually added or deleted. This allows us to perform the same analysis as was done for particle advection without SA.

By visualizing the throughput of our *Baseline* implementation in the roofline model, we see that its performance is rather poor. In order to expose more spatial locality, the particle advection kernel without SA did not access particles indirectly through the *active particles* list, but it processed them directly from the *particles* list. Unfortunately, this solution does not work with self-adaptiveness as active particles might not form a contiguous block of memory anymore. As particles are added and removed, the *particles* list becomes fragmented. As a result, the only way to access active particles is by indirection through the *active particles* list. Spatial locality can still be improved however by sorting the indices in the *active particles* list before launching the particle advection kernel. This is again easily achieved through the *sort()* function in the Thrust library. As the *SortedPtr* graph in Figure 18a shows, this leads to a reduction of the running time. Obviously, such a solution is only useful when the cost of the additional sorting step is no greater than the obtained speedup. The *SortedPtr** graph in Figure 18a

therefore represents the sum of the particle advection kernel and additional sorting step, validating this solution.

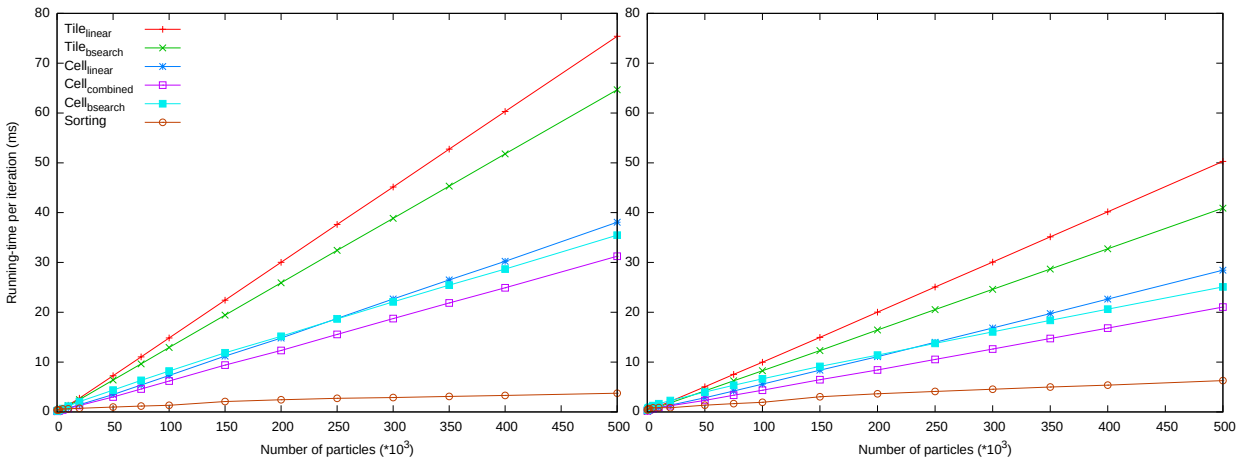
Although sorting the *active particles* list improves running-time, a throughput of only roughly 34 GFLOP/sec is achieved. Unfortunately this is well below the, already pessimistic, $BW_{\texttt{spill}}$ ceiling. We therefore perform an experiment in which the second-order Runge-Kutta scheme is replaced by a first-order Euler step. As Figure 18a(*Euler*) shows, running-times improve. However, because the number of floating-point instructions is halved, and running-times have only decreased by approximately 20%, throughput does not improve. Alternatively, by disabling the evolution of the shape tensor **T**, the number of floating-point instructions is reduced to 631. Additionally, the number of registers used is reduced to 50 and no spilling occurs anymore. This again reduces running-time as shown in Figure 18(*NoTensor*), but due to a lower number of floating-point instructions per particle, throughput does not improve. It therefore appears that the particle advection kernel with self-adaptiveness is not compute- but memory-bound. Unfortunately, it is unclear at this moment what the limiting factor that causes this memory-bound is.

7.5 LEVEL-SET CORRECTION

In Section 6.2, a naive, tile-based and cell-based approach for the level-set correction stage have been discussed. Additionally, a few variants of the tile- and cell-based approach were described that differ in the searching-method used when advancing the particle stencil.

Before discussing the results of these different approaches for correction and their variants, we first consider the effects of indirectly accessing particles and the resulting scattered memory access patterns. Recall that, for the correction of a grid point, threads consider particles that have positions which are spatially close to the grid point and therefore also close to each other. This is achieved by accessing particles through the *active particles* list, after it was sorted. As a result, a thread considers particle-pointers that are close to each other in the *active particles* list. Unfortunately, this locality is lost for the larger particle structures because of indirection. From the results in the previous section, we know that locality in accessing the particles has a significant impact on performance.

Recall that, in Chapter 6, one of the reasons for introducing indirection through the *active particles* list was to avoid sorting of the large particle data structures. When no self-adaptiveness is performed, we can therefore in fact avoid indirection by sorting the particles themselves instead of the pointers in the *active particles* list. Figure 19 presents the results of an experiment in which particle correction is performed using indirection through the *active particles* list in 19a and using direct access to sorted particles in 19b.



(a) With indirection through the sorted *active particles* list.

(b) With sorting of particles and direct access.

Figure 19: Running-times of various methods and variants of the correction kernel without SA.

For a fair comparison, the running-times of the sorting step are included in both graphs. This shows that sorting of particles is indeed as much as 2× more expensive then sorting the *active particles*

list. Sorting is however far less expensive than even the fastest correction method. The sum of the running-times at $500 \cdot 10^3$ particles for the cell-based approach with combined searching method ($Cell_{combined}$) and the *Sorting*, is approximately $31 + 3 = 34\,ms$ when using indirection, as shown in Figure 19a. With sorting and direct access of particles, Figure 19b shows that a combined running-time of $21 + 6 = 27\,ms$ is achieved. In an application in which self-adaptiveness is not required it is therefore beneficial to discard indirection and sort particles directly even with the increased running-time of the intermediate sorting step.

Through its particle management stage, the self-adaptive variant *does* rely on indirection. There is no difference however between running-times of a self-adaptive variant and those reported for the non-SA variant in Figure 19a. This is because the correction process, as indicated in Chapter 6, only uses a particle's position and normal.

Although the cell-based approach requires tracking a much larger stencil, the reduced number of particles to be considered for correction appears to be favourable. This can be seen in Figure 19a as all variants of the cell-based approach have a significantly smaller running-time. The best performing variant uses the combined linear- and binary searching method described in Chapter 6 with a section-size of $Z = 32$.

To understand why this combined method performs better and why a section-size of 32 appears optimal, we first instrument the linear variant such that it reports the number of steps required when advancing each pointer of the stencil.

Data obtained from a test using SA shows that in as much as 84% of all cases, a pointer does not need to be advanced at all. This is the case when a pointer already points to the target particle or a particle having a larger coordinate. A histogram, presented in Figure 20, shows the frequency of occurrence for each of the numbers of steps. This histogram is strongly skewed to the right which can be interpreted as most searches taking relatively few steps.

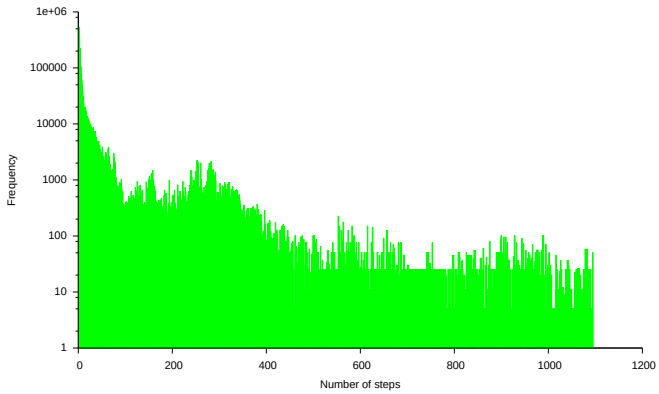


Figure 20: Histogram showing frequencies for the number of steps needed to linearly advance a pointer in the particle-stencil.

From the data, we find that the average number of steps needed to satisfy the linear search is approximately 32. This indicates that a section-size of 64 would be optimal as it considers a particle at a relative offset of 32 in its first step. The difference in running-time between a section-size of 32 and 64 are negligible.

By instrumenting the combined variant, we are able to observe the number of binary-search-steps needed before the search is satisfied. From collected data, the average number of steps needed for a section-size of 32 is 8.6. For a section-size of 64, this is slightly better at 7.8 steps. The difference between these average numbers of steps is relatively small, which is why no significant difference between their running-times was observed. Both present an obvious improvement over the average of 32 steps needed in the linear variant.

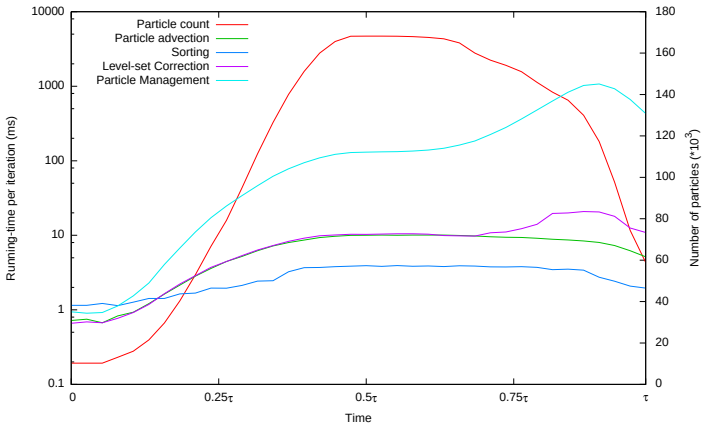
7.6 PARTICLE MANAGEMENT

Running the Enright test with self-adaptiveness, using the parameters listed in Table 7 and recording the average running-times per 100 iterations, allows us to obtain the results presented in Figure 21a.

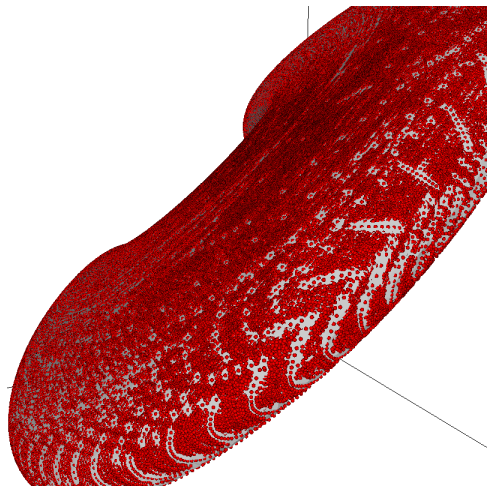
Initial number of Particles	σ	β	d_{min}	α_{min}
10242	1.9	$\frac{2}{3}$	$\frac{1}{3}$	0.5

Table 7: Parameters used for the Enright test with Self-adaptiveness.

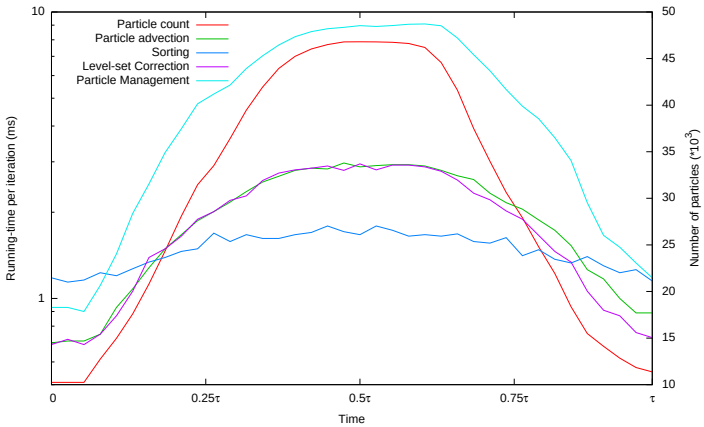
The number of particles increases from 10242 to approximately $170 \cdot 10^3$ at $t = 0.5\tau$, after which the interface starts to contract and de-seeding causes this number to decrease again. In the same figure, the running-times per iteration are shown for each of the new stages that together enable the hybrid approach. At $t \approx 0.75\tau$, although the number of particles starts to decrease, the running-times of the level-set correction- and particle management stages begin to increase, see Figure 21a. These results are explained by considering a snapshot of the interface and particles at $t \approx 0.75\tau$, given in Figure 21b. This snapshot shows how de-seeders have removed particles in their vicinity. However, areas having a high particle density appear to form in between these de-seeders. This invalidates the assumptions on the distribution and density of particles made in Chapter 6, resulting in non-linear behavior in the running-times of the level-set correction- and particle management stages.



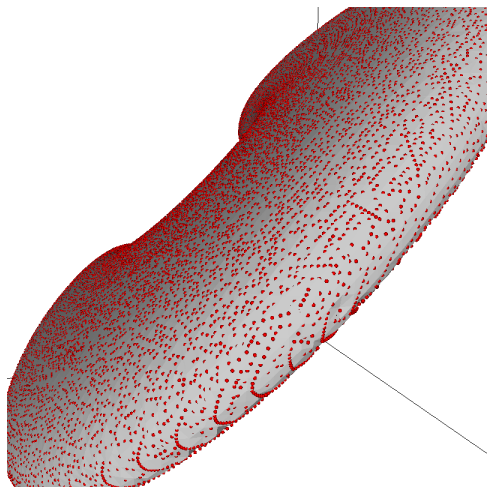
(a) Particle count and running-times during the Enright test using the original de-seeding criterion.



(b) A snapshot of the interface and particles at $t \approx 0.75\tau$ using the original de-seeding criterion.



(c) Particle count and running-times during the Enright test using a less strict de-seeding criterion.



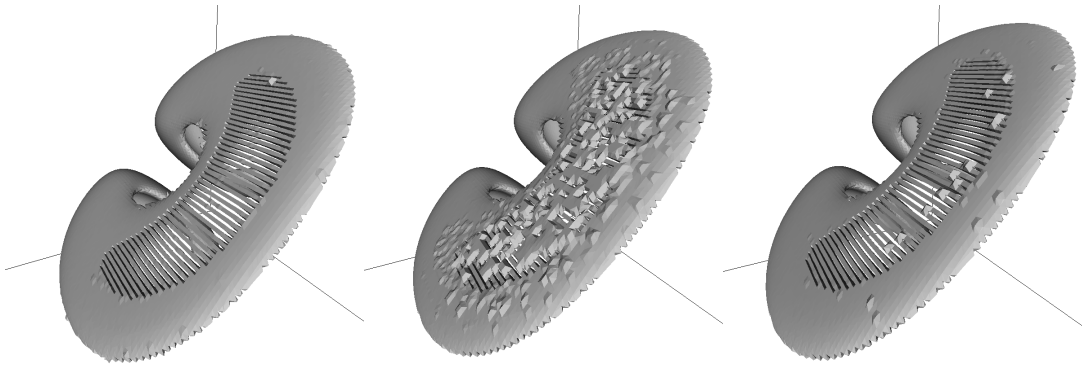
(d) A snapshot of the interface and particles at $t \approx 0.75\tau$ using a less strict de-seeding criterion.

Figure 21: Running-times of stages related to particles and snapshots using the original de-seeding criterion in *(a)* and *(b)*, and a less strict criterion in which all particles are de-seeders in *(c)* and *(d)*. Note the scale differences in *(a)* and *(b)*.

To solve this issue, we enforce a less strict de-seeding condition by allowing each particle to become a de-seeder while still protecting particles initialized at t_0 from being deleted. In theory, as described in Section 4.3.2, this might cause particles that represent less reliable information to be kept in favour of more reliable ones. Particles that were initialized at t_0 and represent the most reliable information are protected however, and in practice there appears to be no noticeable degradation of the results. Because all particles are now de-seeders, cluttering in areas between the original de-seeders no longer occurs and a much more uniform distribution of particles over the interface is maintained. This can be visually verified by comparing the particle distributions in Figure 21b and Figure 21d. From Figure 21c we find that, because the assumptions made in Chapter 6 hold again, i.e., particles are uniformly distributed and there is a relatively low and bounded number of particles per cell, running-times now appear to increase and decrease linearly with the number of particles.

7.7 OVERVIEW

Throughout many of the discussions in this and the previous chapter, a distinction was made between an implementation with and without self-adaptiveness. To give a fair comparison between these two variants, we first try to find the number of particles that allows the non-SA variant to achieve similar results with respect to mass-preservation and appearance.



(a) With Self-adaptiveness (10242 to 46856 Particles) Total running-time: 44.2s.

(b) Without Self-adaptiveness ($100 \cdot 10^3$ Particles) Total running-time: 29.4s.

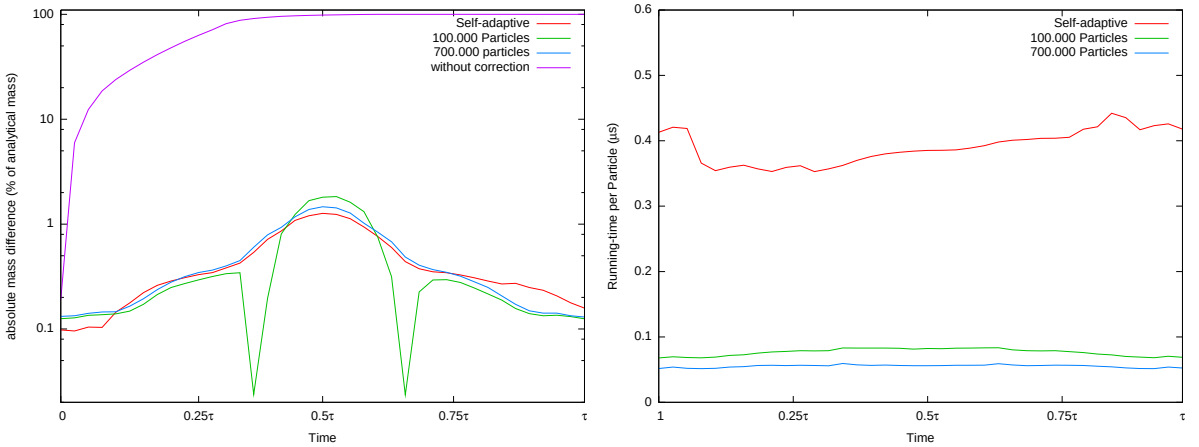
(c) Without Self-adaptiveness ($700 \cdot 10^3$ Particles) Total running-time: 148.6s.

Figure 22: Snapshots of the Enright test with and without SA at $t \approx 0.5\tau$. Note that the interface breaks as a result of aliasing since the interface is too thin to be resolved at a grid size of 128^3.

Results of the variant using SA are presented in Figure 22a and Figure 23. A non-SA implementation with $100 \cdot 10^3$ particles, which is approximately twice as many as the peak number of particles reached in the SA case, results in a reasonable performance with respect to mass-preservation, as shown in Figure 23a. Visually however, i.e., by comparing Figure 22b with 22a, there appears to be a considerable difference. This explains the dips in the graph in Figure 23a(100.000 *Particles*), which are caused by a change of sign since mass is not lost but gained at $t = 0.5\tau$. By increasing the number of particles to $700 \cdot 10^3$, results that are comparable to non-SA, both visually and with respect to mass-preservation, are obtained. Unfortunately, due to the large amount of stretch, most added particles uselessly gather in the areas of high curvature, depleting the flat region visible in Figure 22 of particles. With this knowledge, one might consider changing the initial seeding of particles such that a higher initial density is achieved in areas that will incur large amounts of stretch. Obviously, in most practical situations such knowledge is not available beforehand.

To appreciate the mass-preserving properties of this hybrid approach, consider Figure 24 in which snapshots of the Enright test at $t \approx 0.5\tau$ are shown that were obtained using a non-hybrid, i.e., level-set only, method. Even though this method used a 5th order HJ-WENO scheme for spatial derivatives, a 3rd order TVD Runge-Kutta scheme for time integration and more reinitialization, a mass-loss of 17% occurred.

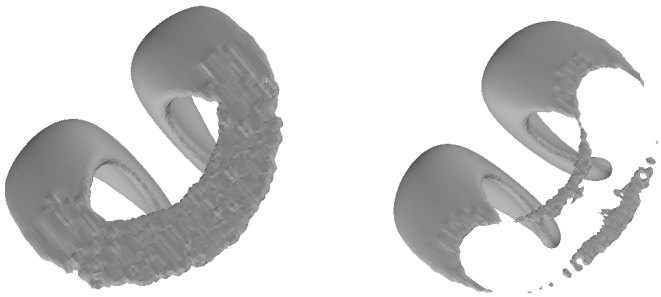
In Section 7.4 and Section 7.5, self-adaptiveness was found to introduce some overhead in the particle advection and level-set correction stages. Together with the relatively expensive particle management



(a) Absolute mass preservation with and without SA corresponding to the results shown in Figure 22.

(b) Total running-time per particle per iteration with and without SA corresponding to the results shown in Figure 22

Figure 23: Results of mass-preservation and running-time per particle for the SA and non-SA variants.



(a) More reinitialization, 17% mass-loss.

(b) Less reinitialization, even more mass is lost.

Figure 24: Results of Enright test at $t \approx 0.5\tau$ with a non-hybrid, i.e., level-set only, method using a 5th order HJ-WENO scheme for spatial derivatives and 3rd order TVD Runge-Kutta scheme for time integration.

stage, as shown in the previous section, it is obvious that self-adaptiveness comes at the cost of a significant amount of overhead. Consider the total running-times reported in Figure 22 for example. The non-SA variant with $100 \cdot 10^3$ particles, having more than twice as many as the maximum number of particles reached using SA, completes its task in 30% less time. This difference can also be observed in Figure 23b, which shows that the running-time per particle with SA is much larger. Compared to the non-SA variant with $700 \cdot 10^3$ particles, which obtains similar results with respect to mass-preservation but uses a much larger number of particles, the SA variant is now faster, completing the task in less than one-third of the time it takes the variant without SA.

It is important to note that the running-times per particle, reported in Figure 23b, remain approximately constant throughout the test. For the non-SA variant, this indicates that the distribution of particles over the interface, which becomes non-uniform as the interface stretches, has little effect on the running-time per particle. For the SA variant, the results support the conclusion made in the previous section that running-times increase linearly with the number of particles. This indicates that the self-adaptive variant is work-efficient.

It might seem strange that, in Figure 23b, the running-time per particle is smaller for a larger number of particles. This can be explained however by considering the results found in Figure 16b which show that, at $100 \cdot 10^3$ particles, the GPU is not yet able to achieve its maximum throughput.

An insight into the relative running-times of all stages in our implementation is provided through Figure 25. This shows that all stages inherited from the GPU-STL implementation, i.e. *Level-set advection*,

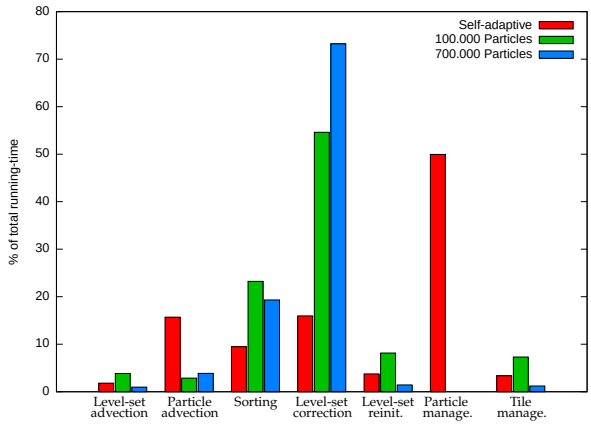


Figure 25: Relative running-time per stage with and without SA.

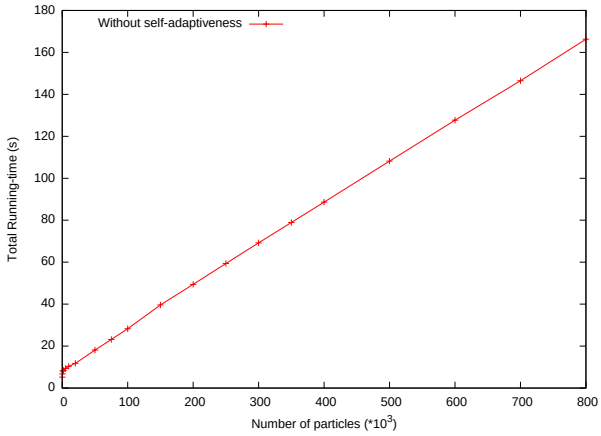


Figure 26: Total running-times of the Enright test without self-adaptiveness for a varying number of particles.

Level-set reinitialization and *Tile management*, only account for a small portion of the running-time. For the implementation that uses self-adaptiveness, 50% of the running-time is spent on *Particle management*. Without self-adaptiveness, the *Level-set correction* stage occupies most of the running-time.

Finally, Figure 26 shows that the total running-time of the non-SA variant, like its SA counterpart, is linear in the number of particles. This indicates that the non-SA variant is also work-efficient.

7.8 COMPARISON TO RELATED WORK

We now compare the results of our hybrid method and its implementation to those of GPU-accelerated implementations found in related work, which were introduced in Section 2.2.

In their poster-presentation[37], Zheng and Enright report results of the Enright test using their GPU-accelerated 'speedy PLS' implementation. Unfortunately, being a poster-presentation, relatively few details are given. A total running-time of 683.71s is reported using a 200^3 grid on an NVIDIA GTX 480. Although it shares the same Fermi architecture with our GTS 450, the GTX 480 is much more powerful. Whereas the GTS 450 has 192 CUDA-cores and a peak memory bandwidth of 57 GB/sec, the GTX 480 has 448 CUDA-cores and a peak memory bandwidth of 177 GB/sec. We therefore assume the GTX 480 to be at least 2.5× faster. Running the Enright test with our self-adaptive implementation on a 256^3 grid with 40962 initially placed particles, results in visually comparable results. The peak particle count is 186412 and it takes 317.41s to complete the test. Correcting this result for difference in speed between the GTX 480 and GTS 450, we estimate that our implementation is at least 5× more efficient.

The PLS implementation of Matsuda et al., described in [18], is based on the OpenGL API and relies on programmable shaders to perform computations. Using a GeForce 8800 Ultra, a single time-step is reported to take 132ms for a 128^3 computational grid. The GeForce 8800 Ultra, based on one of the first CUDA-capable architectures, contains 128 CUDA-cores and has a peak memory bandwidth of 103.7 GB/sec. Although it contains less CUDA-cores when compared to our GTS 450, its peak memory bandwidth is much higher. We therefore consider them to be comparable. Recall that the running-time of a single step in our self-adaptive implementation varies as it depends on the number of particles. At the peak number of particles, a running-time of 18ms is obtained. Our implementation is therefore considered to be at least 7× more efficient.

For their implementation of the PLS method[7], also based on graphics APIs and shaders,Cuntz et al. unfortunately do not report results for the Enright test. Instead, running-times per time-step are reported for a 3D version of Zalesak's disk test on a 128^3 grid. As shown in Section 3.2, Zalesak's disk test does not contain any stretching of the interface and self-adaptiveness would therefore not be necessary. In order still to be able to provide a comparison, we run the Enright test without self-adaptiveness using the same number of 262144 particles as reported in [7]. Note that, based on the results from Chapter 3, we know that our SA-OPLS based method would actually require less

particles to obtain comparable results. Our test results in a running-time of approximately 17ms per time-step. Using a GeForce 8800 GTS, Cuntz et al. report a running-time of 56.7ms per time-step on their test. Because the GeForce 8800 GTS has a comparable peak memory-bandwidth but only carries 128 CUDA-cores, we estimate it to be 35% slower when compared to our GTS 450. After correcting for this difference in speed, we estimate our method and implementation to be approximately 2× more efficient. In their paper, Cuntz et al. also report individual running-times for each of the stages in the algorithm. From these results it appears that level-set reinitialization accounts for almost 70% of the running-time while in our case this is less than 5%.

In [19], Mei et al. present results for their GPU-accelerated implementation of the MLS method. Besides running-times, Mei et al. also report results with respect to mass-preservation for the Enright test on a 128^3 grid. Using 97708 particles, a 1.8% mass-loss is recorded. As results for two-dimensional tests in Section 3.2 have shown, the SA-OPLS method provides comparable mass-preservation results. For the Enright test, using self-adaptiveness, we indeed obtain a comparable peak mass-loss of 1.27%, but with only 46856 particles. With respect to running-times, Mei et al. report obtained framerates. Fortunately, the amount of time needed for rendering, which we did not include in our measurements, is also reported. After correcting the reported 24 FPS for rendering, the running-time per time-step is 37.9ms. This result is obtained using a GeForce 8800 GTX which, like the GTS and Ultra versions, carries 128 CUDA-cores. With its peak memory bandwidth of 86.4 GB/sec, we estimate it to be only 15% slower when compared to our GTS 450. After correcting for the difference in speed between the GTS 450 and 8800 GTX, we estimate that our implementation is approximately 1.8× more efficient. Similar to the PLS implementation of Cuntz et al., level-set reinitialization is reported to account for a substantial part of the running-time.

Finally, we remark that running-times for a CPU implementation of the SA-OPLS method reported by Ianniello and Mascio[11], using a much simpler test with less particles, are close to 400ms per time-step. Recall that the self-adaptive variant of our implementation, for the Enright test, achieved a peak running-time of 18ms. This indicates that our GPU-based implementation is at least one order of magnitude faster.

In the next and final chapter, we conclude the work and results presented in this and previous chapters.

8

CONCLUSIONS AND FUTURE WORK

Through a comparison of a number of different hybrid variants proposed in literature, we have identified the SA-OPLS method to provide excellent mass-preserving properties as well as a robust self-adaptive mechanism. Combined with its relatively simple correction procedure, this made it an ideal candidate for our implementation of a hybrid level-set method on the massively parallel GPU architecture.

The GPU-STL method, with its efficient GPU-accelerated level-set implementation, provided a solid foundation for our hybrid implementation. Many of the core concepts of GPU-STL, such as *sequential iteration with stencil*, have been re-used in the implementation of the particle-related stages.

In our implementation, self-adaptiveness was found to add a considerable amount of overhead. For certain applications, in which interface-stretch is limited, self-adaptiveness might not be necessary. A non-adaptive version can deliver a higher performance in such cases. In the Enright test however, the interface locally undergoes a large amount of stretch. The non-adaptive variant required over $14\times$ more particles to achieve similar results with respect to mass-preservation when compared to the self-adaptive variant. For this reason, the self-adaptive variant provided the best performance.

With respect to scalability, the non-adaptive variant showed linear behavior in its running-time with increasing numbers of particles. The de-seeding strategy of SA-OPLS was found to lead to a non-uniform distribution of particles on the interface, causing the particle management stage to exhibit quadratic behavior in its running-time. A slightly less strict de-seeding strategy was found to resolve this issue, resulting in a more uniform distribution of particles which allowed the running-time of the self-adaptive variant to behave linear in the number of particles. Since both variants are linear in the number of particles, this indicates that our implementation is work-efficient.

The performance results of our implementation compare favourably against other GPU-accelerated hybrid implementations that were reported in recent related work. We estimate that our implementation is $1.8\times$ to $7\times$ faster on comparable tests while providing similar results. Our impression is that much performance could still be gained from optimizations that reduce the overhead of self-adaptiveness.

In their description of the SA-OPLS method, Ianniello and Mascio do not consider using the additional quantities that allow for a higher-order local approximation of the interface, in the level-set correction procedure. While this might intuitively be expected to result in better mass-preservation, we have not been able to obtain results that support this.

In future work, we would like to demonstrate the capabilities of our implementation in a practical use-case. Additionally, we are interested in exploiting the additional information tracked by each particle which provides a rather complete description of local geometry.

BIBLIOGRAPHY

[1] *CUDA C Programming Guide*. NVIDIA. URL http://developer.download.nvidia.com/compute/cuda/4_0/toolkit/docs/CUDA_C_Programming_Guide.pdf.

[2] cuobjdump Application Note. URL http://www.dahlsys.com/upload/cuobjdump.pdf.

[3] CUDA Occupancy Calculator. URL http://developer.download.nvidia.com/compute/DevZone/docs/html/C/tools/CUDA_Occupancy_Calculator.xls.

[4] David Adalsteinsson and James A. Sethian. A fast level set method for propagating interfaces. *J. Comput. Phys.*, 118:269–277, May 1995.

[5] John B Bell, Phillip Colella, and Harland M Glaz. A second-order projection method for the incompressible navier-stokes equations. *Journal of Computational Physics*, 85(2):257 – 283, 1989.

[6] Nicolas Cuntz, Robert Strzodka, and Andreas Kolb. Real-time particle level sets with application to flow visualization. Technical report, 2007.

[7] Nicolas Cuntz, Andreas Kolb, Robert Strzodka, and Daniel Weiskopf. Particle level set advection for the interactive visualization of unsteady 3D flow. *Computer Graphics Forum*, 27(3):719–726, May 2008.

[8] Douglas Enright, Ronald Fedkiw, Joel Ferziger, and Ian Mitchell. A hybrid particle level set method for improved interface capturing. *J. Comput. Phys*, 183:83–116, 2002.

[9] Douglas Enright, Frank Losasso, and Ronald Fedkiw. A fast and accurate semi-lagrangian particle level set method. *Computers and Structures*, 83:479–490, 2004.

[10] Nick Foster and Ronald Fedkiw. Practical animation of liquids. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '01, pages 23–30, New York, NY, USA, 2001. ACM. ISBN 1-58113-374-X.

[11] S. Ianniello and A. Di Mascio. A self-adaptive oriented particles level-set method for tracking interfaces. *Journal of Computational Physics*, 229(4):1353 – 1380, 2010.

[12] Andrei Jalba, Wladimir van der Laan, and Jos Roerdink. Real-time sparse level-sets on graphics hardware. unpublished, 2011.

[13] Guang-Shan Jiang and Danping Peng. Weighted eno schemes for hamilton–jacobi equations. 21 (6):2126–2143, 2000.

[14] Aaron Lefohn, Joe Kniss, Charles Hansen, and Ross Whitaker. Interactive deformation and visualization of level set surfaces using graphics hardware. In *Proceedings of IEEE Visualization 2003*, pages 75–82, 2003.

[15] Aaron E. Lefohn, Joe M. Kniss, Charles D. Hansen, and Ross T. Whitaker. A streaming narrow-band algorithm: interactive computation and visualization of level sets. In *ACM SIGGRAPH 2005 Courses*, SIGGRAPH '05, New York, NY, USA, 2005. ACM.

[16] Randall J. LeVeque. High-resolution conservative algorithms for advection in incompressible flow. *SIAM Journal on Numerical Analysis*, 33(2):627–665, 1996.

[17] Frank Losasso, Jerry O. Talton, Nipun Kwatra, and Ron Fedkiw. Two-way coupled SPH and particle level set fluid simulation. *IEEE Transactions on Visualization and Computer Graphics*, 14(4), 2008.

[18] Yasuhiro Matsuda, Yoshinori Dobashi, and Tomoyuki Nishita. Fluid simulation by particle level set method with an efficient dynamic array implementation on gpu. *IIEEJ Image Electronics and Visual Computing Workshop 2007-11*, 2007.

[19] Xing Mei, Philippe Decaudin, Bao-Gang Hu, and Xiaopeng Zhang. Real-time marker level set on GPU. In *International Conference on Cyberworlds, CW '08, September, 2008*, pages 209–216, Hangzhou, Chine, 2008. IEEE.

[20] Paulius Micikevicius. Local memory and register spilling. URL http://developer.download.nvidia.com/CUDA/training/register_spilling.pdf.

[21] V. Mihalef, M. Sussman, and D. Metaxas. The marker level set method: a new approach to computing accurate interfacial dynamics. unpublished. URL http://www.research.rutgers.edu/~mihalef/mls_jcp.pdf.

[22] V. Mihalef, D. Metaxas, and M. Sussman. Textured liquids based on the marker level set. *Computer Graphics Forum*, 26(3):457–466, 2007.

[23] Emud Mokberi and Petros Faloutsos. A particle level set library. URL http://www.magix.ucla.edu/software/levelSetLibrary/.

[24] Michael B. Nielsen and Ken Museth. Dynamic tubular grid: An efficient data structure and algorithms for high resolution level sets. *J. Sci. Comput.*, 26:261–299, March 2006. ISSN 0885-7474.

[25] S. Osher and R. Fedkiw. *Level Set Methods and Dynamic Implicit Surfaces*. Applied mathematical sciences. Springer Verlag, 2003. ISBN 9780387954820.

[26] Stanley Osher and James A. Sethian. Fronts propagating with curvature dependent speed: Algorithms based on hamilton-jacobi formulations. *Journal of Computational Physics*, 79(1):12–49, 1988.

[27] Frederic Pighin, Sanjit Patel, Jonathan Cohen, and Anson Chu. Oriented particle level set for fluid simulation. Poster - Eurographics/ACM SIGGRAPH Symposium on Computer Animation, 2005. URL http://www.jcohen.name/papers/Pighin_Oriented_2005.pdf.

[28] J. A. Sethian. Evolution, implementation, and application of level set and fast marching methods for advancing fronts. *Journal of Computational Physics*, 169(2):503 – 555, 2001.

[29] J.A. Sethian. *Level set methods and fast marching methods: evolving interfaces in computational geometry, fluid mechanics, computer vision, and materials science*. Cambridge monographs on applied and computational mathematics. Cambridge University Press, 1999. ISBN 9780521645577.

[30] John Strain. Semi-lagrangian methods for level set equations. *J. Comput. Phys.*, 151:498–533, May 1999.

[31] R. Strzodka and M. Rumpf. Level set segmentation in graphics hardware. In *Proceedings of IEEE International Conference on Image Processing (ICIP'01)*, 3, pages 1103–1106, 2001.

[32] Mark Sussman, Peter Smereka, and Stanley Osher. A level set approach for computing solutions to incompressible two-phase flow. *Journal of Computational Physics*, 114(1):146 – 159, 1994.

[33] Steven T and Zalesak. Fully multidimensional flux-corrected transport algorithms for fluids. *Journal of Computational Physics*, 31(3):335 – 362, 1979.

[34] Wladimir van der Laan, Andrei Jalba, and Jos Roerdink. A memory and computation efficient sparse level-set method. *Journal of Scientific Computing*, 46:243–264, 2011.

[35] Zhaoyuan Wang, Jianming Yang, and Frederick Stern. An improved particle correction procedure for the particle level set method. *Journal of Computational Physics*, 228(16):5819 – 5837, 2009.

[36] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM*, 52:65–76, April 2009. ISSN 0001-0782.

[37] Wen Zheng and Douglas Enright. Gpu speedy particles and grids (a gpu implementation of the 3d particle level set method). Poster - Stanford ICME Open House Day 2011, 2011. URL http://physbam.stanford.edu/~zhw/paper/GPU_Speedy_PLS.pdf.