UNIVERSIDADE FEDERAL DO RIO DE JANEIRO
INSTITUTO DE MATEMÁTICA
INSTITUTO DE TÉRCIO PACITTI DE APLICAÇÕES E PESQUISAS COMPUTACIONAIS
PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA

LUCAS RIBEIRO DE AZEVEDO

ESTUDO E IMPLEMENTAÇÃO UTILIZANDO GPU COMPUTING DO MÉTODO MSPH

Rio de Janeiro
2013

UNIVERSIDADE FEDERAL DO RIO DE JANEIRO
INSTITUTO DE MATEMÁTICA
INSTITUTO DE TÉRCIO PACITTI DE APLICAÇÕES E PESQUISAS COMPUTACIONAIS
PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA

LUCAS RIBEIRO DE AZEVEDO

ESTUDO E IMPLEMENTAÇÃO UTILIZANDO GPU COMPUTING DO MÉTODO MSPH

Dissertação de Mestrado submetida ao Corpo Docente do Departamento de Ciência da Computação do Instituto de Matemática, e Instituto Tércio Pacitti de Aplicações e Pesquisas Computacionais da Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários para obtenção do título de Mestre em Informática.

Orientador: Marcello Goulart Teixeira

Rio de Janeiro
2013

A994 Azevedo, Lucas Ribeiro de

Estudo e implementação utilizando GPU computing do método mSPH / Lucas Ribeiro de Azevedo. – 2013.

134 f.: il.

Dissertação (Mestrado em Informática) – Universidade Federal do Rio de Janeiro, Instituto de Matemática, Instituto de Tércio Pacitti de Aplicações e Pesquisas Computacionais, Programa de Pós-Graduação em Informática, Rio de Janeiro, 2013.

Orientador: Marcello Goulart Teixeira.

1. MSPH. 2. SPH. 3. Fluxo. 4. Incompressível. 5. GPU. 6. GPGPU. 7. Paralelo. 8. Opencl. – Teses. I. Teixeira, Marcello Goulart (Orient.). II. Universidade Federal do Rio de Janeiro, Instituto de Matemática, Instituto de Tércio Pacitti de Aplicações e Pesquisas Computacionais, Programa de Pós-Graduação em Informática. III. Título

CDD:

LUCAS RIBEIRO DE AZEVEDO

Estudo e implementação utilizando GPU computing do método mSPH

Dissertação de Mestrado submetida ao Corpo Docente do Departamento de Ciência da Computação do Instituto de Matemática, e Instituto Tércio Pacitti de Aplicações e Pesquisas Computacionais da Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários para obtenção do título de Mestre em Informática.

Aprovado em: Rio de Janeiro, _____ de _____ de _____.

Prof. Dr. Marcello Goulart Teixeira (Orientador)

Prof. Dr. Esteban Walter Gonzalez Clua

Profa. Dra. Juliana Vianna Valerio

Prof. Dr. Thomas Maurice Lewiner

Rio de Janeiro
2013

Aos gênios Charles Babbage, Alan Turing e Gottfried Wilhelm von Leibniz.

AGRADECIMENTOS

Primeiramente, gostaria de agradecer a alguém por quem tenho uma profunda admiração e sem ele certamente este projeto não existiria. Guerreiro, mesmo com todas as dificuldades não desistiu, pois sabia o tempo todo que chegaria lá, afinal sempre se via vencendo. Obrigado Lucas, sim, eu mesmo.

Gostaria de agradecer ao Prof. Mauro Antonio Rincon, não somente por todo amparo e orientação, mas também por ser o grande responsável pela minha linha acadêmica. Graças ao Prof. Rincon, consegui unir quatro coisas que amo, matemática, computação, ensino e pesquisa, não necessariamente nesta ordem.

Gostaria de agradecer ao Prof. Marcello Goulart também pelo amparo, paciência e orientação. Seu espírito de equipe é simplesmente contagiante, o qual me deu força para seguir em frente.

Gostaria de agradecer também aos professores do Programa de Pós-Graduação em Informática - PPGI/UFRJ. Foi excelente passar os anos de mestrado aprendendo com uma equipe cativante, atenciosa e empenhada em ensinar os alunos. Em especial, gostaria de agradecer aos professores Luziane Mendonça, Juliana Valério e Daniel Alfaro. Ainda no âmbito do PPGI, gostaria de agradecer aos amigos do laboratório LC3.

Minha sobrevivência financeira, durante esta maravilhosa jornada, se deu, primariamente, através de bolsa de estudos. Portanto, queria agradecer às entidade responsáveis pela ajuda financeira, CNPQ (Conselho Nacional de Desenvolvimento Científico e Tecnológico) e Cenpes (Centro de Pesquisas Leopoldo Américo Miguez de Mello).

Quero também agradecer ao meu irmão Neymar. Não, este não é o jogador de futebol, mas para mim é um craque, uma estrela, valeu meu irmão e muito obrigado por tudo que fizeste por mim. Ainda falando do mesmo sangue, quero agradecer ao anjo que apareceu na minha vida. Hoje, este anjo eu chamo de mãe. Muito obrigado Dona Neuza, a senhora foi e continua sendo incrível para mim.

Meus avós também foram e são responsáveis pela minha vitória. Muito obrigado pelo apoio. Aproveito para descer na árvore genealógica e agradecer ao meu irmão Pedro, que esteve sempre ao meu lado.

Neste ponto gostaria de agradecer aos meus amigos. Na verdade são tão amigos que ficam na categoria de irmãos. Obrigado pelos momentos compartilhados durante esta minha empreitada, sem a amizade de vocês este passo a mais na minha carreira acadêmica seria muito mais complicado.

E por último, todavia não menos importante, queria agradecer a minha companheira, amiga e meu amor. Karoline, muito obrigado por todo apoio.

RESUMO

Azevedo, Lucas Ribeiro de . **Estudo e implementação utilizando GPU computing do método mSPH**. 2013. 134 f. Dissertação (Mestrado em Informática) - PPGI, Instituto de Matemática, Instituto Tércio Pacitti de Aplicações e Pesquisas Computacionais, Universidade Federal do Rio de Janeiro, Rio de Janeiro, 2013.

O SPH (*Smoothed Particle Hydrodynamics*) é um método numérico sem malha e de partículas lagrangians. A questão chave do SPH é a computação de derivadas parciais sem utilizar qualquer grade. Diferentemente do Método dos Elementos Finitos, por exemplo, as derivadas são encontradas diferenciando analiticamente uma fórmula de interpolação radial conhecida como KI (*Kernel Interpolation*). O SPH foi concebido para simular, inicialmente, modelos estelares politrópicos, i.e. fluxos compressíveis e ilimitados. Posteriormente, o SPH foi alvo de inúmeras extensões e aplicações em diferentes áreas, tais como fluxos incompressíveis, com superfície livre, altamente compressíveis, explosões sub-aquáticas e muito mais.

A técnica de interpolação KI é baseada em busca de vizinhos e sua implementação computacional pode se dar através de um algoritmo de CD (*Collision Detection*). A tarefa do CD é reportar pares de objetos virtuais que, em um instante de tempo, se intersectam em um ambiente também virtual. A etapa de CD em um simulador virtual pode facilmente se tornar o gargalo no tempo de simulação. Recentemente, um algoritmo de CD feito para rodar inteiramente em GPU (*Graphics Processing Unit*) se mostrou 71 vezes mais rápido que uma implementação considerada o estado da arte, esta última utilizando somente a CPU (*Central Processing Unit*). Este é somente um exemplo exibindo a superioridade da GPU, ao efetuar cálculos de natureza paralela, em relação à CPU. Pode-se encontrar inúmeras outras aplicações que igualmente demonstram esta superioridade.

Um dos dois objetivos desta dissertação é exibir o método, recentemente apresentado à comunidade científica, mSPH (*modified SPH*), no contexto original para o qual foi criado, i.e., simulação de fluxos incompressíveis e não lineares. Este método nasceu após a primeira extensiva análise numérica e analítica do SPH e dos tratamentos semi-empíricos aplicados ao mesmo. Esta análise mostrou que o SPH é inerentemente instável, todavia passível de regularização. Graças a essa análise, segundo a autora, o mSPH é um método convergente, utilizando condição de Courant, sem qualquer tratamento semi-empírico.

O outro objetivo desta dissertação é fornecer uma implementação computacional eficiente do mSPH, esta última feita para rodar em paralelo na GPU. Conhecendo o nível de dificuldade encontrado no uso da GPU para efetuar cálculos de propósito

geral, optou-se por detalhar a implementação em diferentes níveis de abstração. Uma atenção especial é dada à etapa de CD (*Collision Detection*), e sua implementação é baseada no algoritmo referenciado acima. Até o momento, esta é a primeira vez que uma implementação desse algoritmo é apresentada detalhadamente.

Palavras-chave: MSPH, SPH, fluxo, incompressível, GPU, GPGPU, paralelo, opencl.

ABSTRACT

Azevedo, Lucas Ribeiro de . **Estudo e implementação utilizando GPU computing do método mSPH**. 2013. 134 f. Dissertação (Mestrado em Informática) - PPGI, Instituto de Matemática, Instituto Tércio Pacitti, Universidade Federal do Rio de Janeiro, Rio de Janeiro, 2013.

The SPH (Smoothed Particle Hydrodynamics) is a numerical, Lagrangian and meshless particle method. It does not relies upon any grid, as Finite Element Method (FEM) does for example, to calculate spatial derivatives. Instead, they are found by analytical differentiation of interpolation formulae known as KI (Kernel Interpolation). The SPH was initially created to simulate polytropic stellar models, i.e., unbounded, compressible flows. But since then, it has been extended and applied to a wide variety of problems, including incompressible flows, free surface flows, high compressible flows and underwater explosion, just to name few.

The KI implementation relies on neighbors search algorithm and this last one can be seen as a CD (Collision Detection) algorithm. The CD task is responsible for reporting pairs of virtual objects that, for a given time step, have their geometries intersecting among themselves in a virtual environment. The CD step can easily, given a virtual simulator, becomes the simulation time bottleneck. Recently, a GPU-based CD algorithm presented showed to be 71 times faster than a considered state-of-art CPU-based CD. This is just a example showing how much superior GPU can be, doing parallel computation, in relation to CPU, although many more examples can be found.

One of the two goals of this dissertation is to exhibit the method, recently presented to the scientific community, mSPH (modified SPH), within the original context that it war create for, i.e., incompressible and non-linear flows simulation. This method was born right after the first extensive analytical and numerical investigation of the SPH for free-surface flows, and the common semi-empirical treatments applied to it. The investigation not only showed that the SPH method is inherently unstable, but also concluded that a numerical regularization could be applied. The analysis findings allowed the suggestion of a modified SPH scheme that removes the SPH semi-empirical treatments and, according to the author, is stable with known convergence properties bases on Courant condition.

The second goal is to present a efficient computational implementation of the mSPH method, that runs parallel on GPU. Knowing how difficult is to use the GPU for general computation purpose, it was decided to detail the implementation in several abstraction levels. A special attention is given to the CD algorithm, and it's

implementation is based on the algorithm referenced above. As far as it's known, this is the first time to present this kind of implementation with so many details.

Keywords: mSPH, SPH, flow, incompressible, GPU, GPGPU, parallel, opencl.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

EDO	Equação Diferencial Ordinária
SPH	*Smoothed Particle Hydrodynamics*
MA-SPH	*Main Algorithm SPH*
CMA-SPH	*Continuous Main Algorithm SPH*
mSPH	*Modified SPH*
XSPH	X expressa um fator desconhecido, ver [38]
KI	*Kernel Interpolation*
TSE	Tratamentos Semi-Empíricos
FAO	Frequência Altamente Oscilatória
MLS	*Medium Least Square*
CPU	*Central Processing Unit*
GPU	*Graphics Processing Unit*
GPGPU	*General-Purpose computing on GPU*
SAP	*Sweep And Prune*
CD	*Collision Detection*
BP	*Broad Phase*
BV	*Bounding Volume*
NP	*Narrow Phase*
SE	Subdivisão Espacial
CD-SPH	Algoritmo de detecção de colisão implementado nesta dissertação
SM	*Stream Multiprocessor*
ALU	*Arithmetic Logic Unit*
SIMD	*Single Instruction Multiple Data*
SPMD	*Single Program Multiple Data*
RAM	*Random Access Memory*

AoS	*Array-of-Structures*
SoA	*Structure-of-Arrays*
VLIW4	*Very Long Instruction Word*

LISTA DE ALGORITMOS

SUMÁRIO

1 INTRODUÇÃO

1.1 Motivação

O estudo da dinâmica de fluidos, em particular os violentos e com superfície livre, mostra-se importante para a compreensão de fenômenos, tais como a evolução dos litorais [32] e o aumento da transferência de gás, calor e energia entre o oceano e a atmosfera [33].

Fluxos violentos e com superfície livre são pouco entendidos, pois são altamente não lineares, transientes, multi-escala, bifásicos e envolvem simultaneamente múltiplas regiões conectadas de água e ar. Mostrando-se portanto praticamente intratáveis pela matemática analítica. Além disso, os experimentos físicos são difíceis de serem reproduzidos, oscilam bastante devido à compressibilidade do ar e sofrem influências de origens desconhecidas em escalas espaciais e temporais [22].

Com o advento da computação de alto desempenho, os métodos numéricos estão se estabelecendo como uma ferramenta indispensável nos estudos do campo da hidrodinâmica. Os métodos existentes para a resolução de fluxos com superfície livre em macroescala já estão consolidados [7], porém os métodos numéricos para simular fenômenos complexos em mesoescala, tais como spray de jet, formação de gotas, quedas-d'água e rompimento de represas, são assuntos ativos de pesquisas. O desenvolvimento de métodos de simulação em mesoescala é importante para o entendimento dos fenômenos nesta escala e, eventualmente, para complementar os métodos em macroescala para a simulação de fluxos com superfície livre.

Os métodos de simulação em mesoescala devem ser robustos o suficiente para simular a complexidade arbitrária das superfícies livres. Entre os métodos sem malha, o método de simulação numérico SPH (*Smoothed Particle Hydrodynamics*) [19, 29] se mostrou capaz de simular, de forma robusta, fluxos com superfície livre de complexidade arbitrária. Além disso, é relativamente simples de programar e

muito eficiente [26]. Estas características tornaram o método popular, contribuindo para a sua utilização nas simulações em mesoescala, em diferentes áreas [26], incluindo astrofísica, hidrodinâmica, magneto-hidrodinâmica, explosão gasosa e fluxo granular.

O método SPH foi concebido para resolver, inicialmente, problemas no campo da astrofísica, aplicado a modelos estelares politrópicos, isto é, fluxos compressíveis, ilimitados e não viscosos, sob a influência de forças de corpo, tais como gravitacional, rotacional e magnética [19, 29].

Por ser um método sem malha, no SPH as derivadas espaciais não são computadas usando-se uma grade, mas uma técnica de diferenciação analítica das fórmulas de interpolação. Estas fórmulas vêm da teoria de Integrais Interpolantes, onde uma função qualquer $A(\mathbf{x})$ pode ser expressa por $A(\mathbf{x}) = \int_\Omega A(\mathbf{s})\delta(\mathbf{x}-\mathbf{s})\,\mathbf{ds}$ onde $\delta(\mathbf{x}-\mathbf{s})$ é a função delta de Dirac e Ω é todo o domínio. Porém, por motivos numéricos, em SPH a função delta é aproximada por uma outra função, $W(\mathbf{x}, h)$, chamada de núcleo, sendo h seu comprimento de suavização, com as propriedades $\int_\Omega W(\mathbf{x}-\mathbf{s}, h)\,\mathbf{ds} = 1$ e $\lim_{h\to 0} W(\mathbf{q}, h) = \delta(\mathbf{q})$. Normalmente se escolhem funções núcleo com suporte compacto, isto é, $W(\mathbf{q}, h) = 0$ se $\|\mathbf{q}\| \geq \kappa h$ (onde κ é uma constante associada a cada função núcleo), limitando-se assim o domínio de influência da função W. Esta técnica é chamada de *Kernel Interpolation* (KI) e desempenha o conceito chave do SPH.

Discretizando-se o domínio em N partículas lagrangeanas, onde cada partícula a possui massa m_a, densidade ρ_a, posição $\mathbf{x}_a$, velocidade $\mathbf{v}_a$ e $a = 1, 2, ..., N$, a forma discretizada do KI é dada por $A(\mathbf{x}_a) = \sum_{b=1}^{N} \frac{m_b}{\rho_b} A(\mathbf{x}_b) W(\mathbf{x}_a - \mathbf{x}_b, h)$ e o cálculo do seu gradiente é dado por $\nabla A(\mathbf{x}_a) = \sum_{b=1}^{N} \frac{m_b}{\rho_b} A(\mathbf{x}_b) \nabla W(\mathbf{x}_a - \mathbf{x}_b, h)$, onde foi usado que $\rho(\mathbf{x}_a)\mathbf{ds} = m(\mathbf{x}_a)$. Deve-se observar que, como W normalmente tem suporte compacto, o somatório do KI envolve somente as partículas dentro do domínio Ω_a.

Posteriormente, o SPH é estendido para fluxos hidrodinâmicos, incompressíveis e com superfície livre, demonstrando robustez em simular fluxos altamente não lineares com superfícies livres complexas [40]. Para manter a simplicidade do algo-

ritmo original [19, 29], o autor elabora três hipóteses: i) introduz uma velocidade do som artificial e uma equação artificial do Estado para, então, dissociar a pressão da velocidade e manter a natureza explícita do algoritmo. Esta é conhecida como hipótese de compressibilidade fraca; ii) estende o KI para o cálculo das derivadas parciais na superfície livre e iii) para modelar a condição de fluxo zero na fronteira, o autor faz uso de forças potenciais encontradas em simulações moleculares do tipo Lennard-Jones, ajustáveis e altamente sensíveis a parâmetros. Embora condições de fronteira de caráter mais físico tenham sido desenvolvidas [43, 45], esta hipótese estabeleceu a natureza empírica do SPH em que vários tratamentos ajustáveis e semi-empíricos, específicos a determinadas aplicações, são desenvolvidos para tentar resolver diversos problemas nas simulações.

Uma característica do método SPH é sua falta de estabilidade. Na tentativa de amenizar tal característica, inúmeros tratamentos semi-empíricos (TSE), com efeitos desconhecidos na física simulada, foram sendo introduzidos no SPH ao longo de sua existência. Diante deste cenário, foi desenvolvida em [22] uma investigação analítica e numérica do método SPH para fluxos com fronteira livre. Para isto, a autora elaborou o MA-SPH (*Main Algorithm SPH*), uma (re)formulação do método SPH sem TSE, permitindo efetuar uma análise quantitativa e unificada do método numérico e da física que ele captura, buscando avaliar a consistência, estabilidade e convergência do método SPH no contexto de fluxos com fronteira livre.

A análise desenvolvida em [22] mostrou que o SPH é incondicionalmente instável na presença de i) superfície livre e ii) gradiente de densidade base não zero, que é uma generalização da instabilidade tensorial e, para fluxos com fronteira livre, a densidade base se reduz à hidrostática. Apesar da conclusão sobre a estabilidade do método, a análise indicava que o método poderia ser regularizado.

De posse do conhecimento obtido através da análise do método, a autora de [22] investigou os efeitos de alguns dos TSE existentes empregados no SPH. Com base nesta investigação e na análise feita no MA-SPH, a autora desenvolveu o mSPH (*modified SPH*), uma variação do MA-SPH, porém robusto e, segundo a

autora, convergente sem qualquer TSE [22].

Pode-se implementar computacionalmente o SPH empregando as etapas a seguir: i) descobrir os pares de partículas interagindo, *i.e.*, para cada partícula de fluido, descobrir suas "partículas vizinhas" (partículas que estão a uma distância $d < \kappa h$ desta partícula); ii) montar as EDOs que governam as variáveis de cada partícula de fluido, usando a informação obtida na tarefa anterior e iii) resolver as EDOs de cada partícula de fluido. A figura 1.1 ilustra tais procedimentos em um fluxograma.

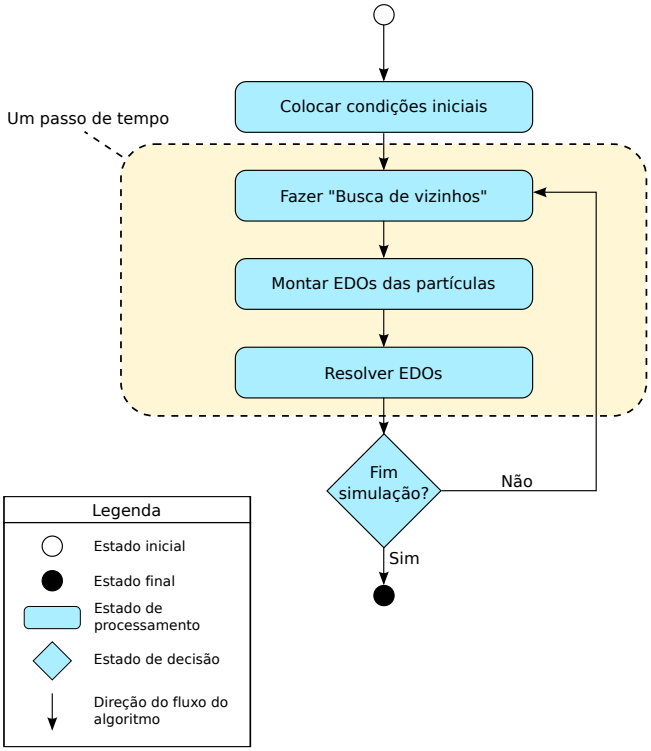

Figura 1.1: Fluxograma de uma implementação do SPH em alto nível.

A tarefa i), da implementação citada no parágrafo anterior, pode ser traduzida em um algoritmo de detecção de colisão [27] (ou em inglês: *Collision Detection* - CD), em particular, de discos (no 2d ou esferas no 3d) com raio $kh/2$. O propósito de um algoritmo de CD é detectar colisões entre objetos em um ambiente virtual

para, posteriormente, por exemplo, impedir que suas geometrias ocupem o mesmo lugar no espaço. No SPH, utiliza-se a informação de saída do algoritmo CD para calcular as derivadas espaciais através do KI. Esta tarefa é o ponto crítico de uma implementação do SPH, no que tange o aspecto tempo de processamento, devido ao custo computacional. Porém, é paralelizável.

Toda partícula de fluido possui um conjunto de EDOs que devem ser montadas a cada iteração do SPH. Esta montagem pode ser feita em paralelo pois não há dependência entre quaisquer variáveis. Logo, a tarefa ii) é paralelizável.

E, por último, as EDOs que governam as partículas de fluido, no SPH, são desacopladas. Portanto, podem ser resolvidas em paralelo, tornando a tarefa iii) também paralelizável assim como as tarefas i) e ii). Isto evidencia o fato de que uma implementação do SPH pode se beneficiar de processamento paralelo.

O fato do SPH ser paralelizável, torna-o candidato a rodar em uma plataforma paralela, tanto de hardware como de software. E, como a GPU (*Graphics Processing Unit*), analisada mais à frente, tem-se mostrado uma grande aliada da computação de algoritmos paralelos, é possível pensar em implementar o SPH para rodar em GPU.

A GPU nasceu para liberar a CPU (*Central Processing Unit*) das tarefas gráficas. Como um processador secundário, a principal tarefa realizada pela GPU era digitalizar uma definição de imagem, dada por uma aplicação, em um conjunto de píxels. Mais tarde, com o avanço do hardware da GPU, esta passou também a ser usada para tarefas não gráficas, *i.e.*, cálculos de propósito geral. Nascia assim a corrente GPGPU (*General-Purpose computing on GPU*) [50] (ou, recentemente, GPU *Computing*).

A computação de propósito geral usando a GPU está baseada em uma arquitetura paralela, tanto de hardware como de software. Pode-se encontrar na literatura vários exemplos que tiram proveito da capacidade de processamento paralelo da GPU usando GPGPU, os quais por vezes demonstram superioridade de performance da GPU sobre a CPU para computação de algoritmos paralelos.

São exemplos de comparação de performance GPUxCPU: i) ordenação de sequência de números [35] (GPU até 3.8x mais rápida); ii) detecção de colisão grosseira (*Broad-phase Collision Detection*) [25] (GPU até 71x mais rápida); iii) primeira implementação do SPH para simulações de fluidos com superfícies livres, com o objetivo de rodar a aplicação de forma interativa [44]. O objetivo foi alcançado ao usar parte da implementação em GPU; iv) implementação completa do SPH em GPU [20] (GPU até 28x mais rápida) e v) resolução de dinâmica de corpos rígidos [46] (GPU até 20x mais rápida).

Portanto, estudar o mSPH com o auxílio do poder computacional da GPU pode resultar em uma combinação sinérgica.

1.2 Escopo da dissertação

O escopo desta dissertação pode ser dividido em três partes: i) é exibido o método MA-SPH, definido em [22], livre de TSE. Este método pode servir como método base para outros métodos SPH, tal como serviu para o mSPH; ii) também definido em [22], o mSPH é um método, de acordo com a autora, convergente, com propriedade de convergência conhecida, capaz de simular fluxos violentos com superficie livre de forma robusta. Este método também é exibido; e iii) pontos chaves de uma implementação eficiente, em termos de tempo de processamento, do SPH são abordados. Estes pontos chaves são válidos para qualquer versão do SPH que utilize o KI, em particular o mSPH. Um exemplo de ponto chave, já citado neste texto, é a busca de vizinhos a uma determinada partícula a, ou seja, partículas que estejam dentro do domínio Ω_a desta partícula a.

Para simulações de métodos sem malha nas quais há colisões entre um grande número de partículas, a etapa de detecção de colisão pode usar grande parte do tempo de simulação. Este fato se agrava no SPH, pois quanto maior o número de partículas vizinhas, melhor a aproximação do método. Portanto, quanto maior o número de colisões, mais acurácia o método terá. Logo, um algoritmo de CD deve ser eficiente. Vê-se então a importância de abordar os aspectos cruciais da

implementação do algoritmo de CD.

1.3 Estrutura da dissertação

Esta dissertação é estruturada em 6 capítulos. No capítulo 2 é descrito o MA-SPH, uma espécie de formulação canônica do SPH, livre de TSE e baseada na formulação SPH apresentada em [40]. Para isso, inicia-se o capítulo descrevendo, brevemente, as duas principais características do método SPH: hipótese de compressibilidade fraca e a técnica de interpolação KI. Depois, um problema canônico é exposto com o intuito de se definir, em seguida, a forma contínua do MA-SPH ou, de forma abreviada, CMA-SPH. Com a formulação contínua do SPH (CMA-SPH) definida, é feita então a discretização desta para elaborar o MA-SPH. E então, abordam-se algumas soluções para as condições de fronteira e, em seguida, a questão da evolução temporal no MA-SPH. Por último, dá-se uma descrição detalhada do KI.

No capítulo 3, a formulação do mSPH é dada, descrevendo-se suas equações governantes, a discretização espacial, as condições de fronteira e iniciais, integração temporal e por último, a regularização do método é exibida.

No começo do capítulo 4, é feito um resumo do paradigma GPGPU contextualizando as características tanto do hardware quanto do software. Logo após, são descritas as primitivas utilizadas para confeccionar a implementação do mSPH em GPU. As primitivas são: i) soma prefixal (*Scan*) que serve como base para o algoritmo de ordenação, entre outras coisas; ii) ordenação *Radix Sorting*, usado na construção do algoritmo de detecção de colisão; iii) detecção de colisão na GPU, necessária para descobrir quais pares de partículas estão interagindo a cada passo de tempo e iv) sublistas em um vetor ordenado com repetição (primitiva chamada nesta dissertação de Sublistas). Uma vez determinados todos os pares de interação, pode-se usar esta primitiva para iterar por estes pares. Ao final do capítulo, é dada uma descrição, baseada em fluxogramas, da implementação do mSPH feita ao longo deste trabalho.

O capítulo 5 tem como objetivo validar a implementação da formulação mSPH. A validação da formulação é baseada no caso de teste padrão o qual simula uma quebra de barragem [58]. O caso de teste permite observar a evolução complexa de um fluxo inicialmente em repouso, com apenas a componente hidrostática na pressão. Após a quebra da barragem (modelada de forma implícita), o fluido percorre um pequena distância horizontal até sofrer um impacto em uma parede vertical. Nesta parede fica um sensor para realizar medições de pressão ao longo do tempo. Na extensão do caminho percorrido pelo fluxo, estão fixados dois sensores para calcular a altura máxima do fluxo, também ao longo do tempo. Os três sensores são responsáveis por coletar dados para comparar resultados de experimentos reais e numéricos.

O último capítulo, o de número 6, é reservado para reportar as conclusões sobre o estudo executado neste trabalho de mestrado, destacando-se as contribuições da dissertação, perspectivas no mSPH, perspectivas na implementação do mSPH em GPU e indicar possíveis rotas para trabalhos futuros.

2 FUNDAMENTOS DO MÉTODO MA-SPH NO CONTEXTO DE FLUIDOS COM SUPERFÍCIE LIVRE

2.1 Introdução

O método SPH foi inicialmente concebido para simular gases interestelares e não limitados [19, 29], sendo portanto uma ferramenta capaz de simular, a priori, fluidos compressíveis e de domínio ilimitado. Em [40] estendeu-se o SPH para simular, por exemplo, quebra de represa, isto é, fluido incompressível e com fronteira livre. Para tal extensão, o autor de [40] fez uso de dois conceitos chave: i) empregou o KI, sem qualquer alteração, para computar as derivadas espaciais, mesmo próximo da fronteira livre onde o KI é incompleto e ii) utilizou a hipótese de compressibilidade fraca no modelo contínuo, tornando o algoritmo explícito, eficiente e simples.

Apesar da atratividade destes dois conceitos, eles introduziram as maiores questões e incertezas com relação à:

- Acurácia - Aparece de forma mais evidente na dinâmica do fluido em que se pode encontrar oscilações de frequência com alta amplitude.

- Consistência - Surge no momento da saída das partículas de suas respectivas posições iniciais ou quando se aproximam da fronteira livre.

- Estabilidade - Eventualmente aparece em simulações de longa duração [48].

Para responder estas questões e incertezas foi elaborado, em [22], o MA-SPH. Este último é o produto da tentativa de se estabelecer uma formulação SPH canônica e livre de TSE para então fazer a análise do método no contexto de fluxos de fluidos com fronteira livre.

No intuito de exibir a formulação do MA-SPH e suas principais caraterísticas, inicia-se na seção 2.2 a descrição da formulação contínua e discreta do MA-SPH, detalhando suas equações constitutivas, hipóteses, condições de fronteira e evolução temporal das variáveis do problema. A seção 2.3 é reservada para detalhar o KI. E por último, em 2.4 comenta-se os resultados obtidos a partir da análise dos TSE feita em [22] utilizando o MA-SPH.

2.2 Formulação do MA-SPH

2.2.1 Descrição do problema

Seja um sistema cartesiano ortogonal $\mathbf{x} = (x, y)$, onde y aponta para cima. Seja também $t > 0$ a variável representando o tempo.

Domínio : O fluxo, inicialmente não perturbado, é limitado na direção vertical $y \in [-H, 0]$ e ilimitado na horizontal. A fronteira livre ∂_{fs} é denotada por $\eta = \eta(x, t)$, onde inicialmente $\eta = y = 0$, já a fronteira fixa, impermeável, ∂_b é dada por $y = -H$.

Funções desconhecidas do fluido: Permita que as funções abaixo representem a velocidade, densidade e pressão do fluxo, respectivamente, no ponto $(\mathbf{x}, t)$.

$$\mathbf{u}(\mathbf{x}, t) = (u, v) \tag{2.1}$$

$$\rho(\mathbf{x}, t) \tag{2.2}$$

$$P(\mathbf{x}, t) \tag{2.3}$$

Definições e hipóteses: Ao menos duas formas de simular fluxos, utilizando-se o método SPH, podem ser encontradas na literatura. Na primeira forma [12] o fluxo é considerado não viscoso e incompressível, resultando nas equações governantes de Euler para o momento e na equação de continuidade para a conservação da massa. Com efeito, a velocidade acopla-se à pressão através da equação de Poisson. Nesta abordagem, a simulação numérica requer o uso de inversão de matriz a cada passo de tempo e faz-se necessário também rastrear a superfície livre.

Na segunda abordagem [6] o fluxo é também considerado não viscoso, porém

isotérmico e fracamente compressível. O fluxo permanece sendo governado pelas equações de Euler para o momento e pela conservação da massa para a densidade, porém a equação de Poisson para a pressão é substituída por uma equação constitutiva do estado. Tal abordagem permite uma simulação numérica mais simples de programar, mais eficiente e que pode ser explícita.

Através de análise dimensional, feita em [40], o autor argumentou que um fluido incompressível pode ser modelado, em uma simulação numérica, como sendo fracamente compressível. O erro desta modelagem é da ordem $O(\frac{1}{c^2})$, sendo garantida desde que o número de *Mach* $M \equiv \frac{|\mathbf{u}|_{\max}}{c} \ll 1$. Por razões de eficiência numérica, determinada pela condição de Courant, a velocidade c não precisa ser a velocidade real do som no fluido, mas uma outra artificial menor possível, *e.g.*, $c \sim 10|\mathbf{u}|_{\max}$.

Duas equações do estado são usadas de forma predominante no SPH [10, 13, 40]. A primeira, e mais usada, é a equação de Tait para gases ideais. Seja ρ_f a densidade não perturbada do fluido e c a velocidade do som artificial, a equação de Tait é

$$P = c^2 \rho_f \left[\left(\frac{\rho}{\rho_f} \right)^{\gamma} - 1 \right] \tag{2.4}$$

onde se admite que o fluido possui razão de compressibilidade γ, onde tipicamente tem-se $\gamma = 7$.

A segunda equação segue da dinâmica de fluidos tradicional [23, 28], relacionando mudanças na pressão a mudanças na densidade com uma equação na forma

$$dP = c^2 d\rho. \tag{2.5}$$

Pode-se mostrar que as duas equações do Estado apresentadas são equivalentes sob a hipótese da compressibilidade fraca [22].

O MA-SPH emprega a segunda abordagem, descrita para simular fluxos fracamente compressíveis. Sendo $\frac{d}{dt}$ a derivada total ou material, *i.e.*, $\frac{d}{dt} \equiv \frac{\partial}{\partial t} + \mathbf{u} \cdot \nabla$ e permita $\mathbf{g} = -g\hat{\mathbf{j}}$ denotar a aceleração da gravidade, as equações governantes para

este fluxo ligeiramente compressível são

$$\frac{d\mathbf{u}}{dt} = -\frac{1}{\rho}\nabla P + \mathbf{g} \tag{2.6}$$

$$\frac{d\rho}{dt} = -\rho\nabla\cdot\mathbf{u} \tag{2.7}$$

$$dP = c^2 d\rho \tag{2.8}$$

a equação do momento de Euler, a conservação da massa para a densidade e a equação do Estado, respectivamente. Estas equações governantes descrevem a forma contínua do MA-SPH, chamada de CMA-SPH.

2.2.2 Discretização espacial

No SPH, método sem malha, o fluxo é discretizado em $N \gg 1$ partículas lagrangianas, onde cada partícula a possui massa constante m_a. Seja $\mathbf{x}_a(t)$, $\mathbf{u}_a(t)$, $\rho_a(t)$ e $P_a(t)$ a posição, a velocidade, a densidade e a pressão, respectivamente, do centro de massa da partícula a no instante de tempo t. As equações para o fluxo discretizado são

$$\frac{d\mathbf{x}_a}{dt} = \mathbf{u}_a \tag{2.9}$$

$$\frac{d\mathbf{u}_a}{dt} = -\frac{1}{\rho_a}\nabla P|_a + \mathbf{g} \tag{2.10}$$

$$\frac{d\rho_a}{dt} = -\rho_a\nabla\cdot\mathbf{u}|_a \tag{2.11}$$

$$P_a = c^2 d\rho_a \tag{2.12}$$

derivadas das equações 2.6, 2.7 e 2.5. Para o cálculo dos termos $\nabla P|_a$ e $\nabla\cdot\mathbf{u}|_a$ no domínio discretizado, usa-se o KI, portanto as equações do MA-SPH são

$$\frac{d\mathbf{x}_a}{dt} = \mathbf{u}_a \tag{2.13}$$

$$\frac{d\mathbf{u}_a}{dt} = -\frac{1}{\rho_a}\sum_{b=1}^{N} P_b\frac{m_b}{\rho_b}\nabla W_{ab} + \mathbf{g} \tag{2.14}$$

$$\frac{d\rho_a}{dt} = -\rho_a\sum_{b=1}^{N}\frac{m_b}{\rho_b}\mathbf{u}_b\cdot\nabla W_{ab} \tag{2.15}$$

$$P_a = c^2(\rho_f - \rho_a) \tag{2.16}$$

onde $\nabla W_{ab} \equiv \nabla W(\mathbf{x}_a - \mathbf{x}_b, h)$.

2.2.3 Condições de fronteira

Para um fluxo não viscoso, com superfície livre e governado por 2.6 e 2.7, as condições de fronteira são:

1. $\mathbf{u} \cdot \hat{n} = 0$, onde $\hat{n}$ é a normal da fronteira não permeável.

2. $P(x, y = \eta, t) = 0$, pressão zero na superfície livre.

3. $\dot{y}_{fs} = \frac{d\eta(x,t)}{dt}$, continuidade cinemática da superfície livre.

No MA-SPH, as condições de fronteira acima são modeladas da seguinte forma, respectivamente:

1. Uso de partículas fantasmas, espelhadas com relação à fronteira [10].

2. Condição não é imposta, pois o KI é incompleto na superfície livre. A estratégia é usar o KI normalmante, gerando erros numéricos, e posteriormente suavizar tais erros. [22].

3. Satisfeita automaticamente, pois em todos os métodos lagrangianos tem-se $\frac{dy_a}{dt} = \frac{d\eta}{dt}$ para $a \in \partial_{fs}$.

2.2.4 Evolução temporal

Para a evolução temporal das funções de campo de uma partícula a, isto é, $\mathbf{x}_a$, $\mathbf{u}_a$ e ρ_a, define-se $q_a^n = [\mathbf{x}_a^n, \mathbf{u}_a^n, \rho_a^n]$ como sendo a posição, a velocidade e a densidade, respectivamente, no tempo t^n. Seja $Q_a^n = [\mathbf{V}_a^n, \mathbf{A}_a^n, R_a^n]$ a taxa de variação no tempo da posição, da velocidade, e da densidade, respectivamente, a quantidade q_a^n é avançada até o próximo passo de tempo $t^{n+1} = t^n + \delta t$ com um esquema de integração apropriado da forma:

$$q_a^{n+1} = q_a^n + Q_a^n \delta t. \tag{2.17}$$

2.3 Definição do KI (*Kernel Interpolation*)

2.3.1 Fundamentos

Basicamente, esta técnica permite qualquer função ser expressa em termos dos seus valores em um conjunto de pontos desordenados, pontos estes chamados, no SPH, de partículas. Seguindo o formalismo apresentado em [39], a base para a construção do KI (*Kernel Interpolation*) segue da teoria de Interpolação Integral. Seja uma função qualquer $A(\mathbf{r})$, sua **Representação Integral**

$$A(\mathbf{x}) = \int_{\Omega} A(\mathbf{x}')\delta(\mathbf{x} - \mathbf{x}')d\mathbf{x}' \tag{2.18}$$

é exata, onde $\delta(\mathbf{x} - \mathbf{x}')$ é a Distribuição de Dirac e Ω é todo o espaço. Agora, permita aproximar $\delta(\mathbf{x} - \mathbf{x}')$ por uma função $W(\mathbf{x} - \mathbf{x}', h)$, chamada no SPH de Núcleo (*Kernel*), possuindo as propriedades

$$\int_{\Omega} W(\mathbf{x} - \mathbf{x}', h) = 1 \tag{2.19}$$

e

$$\lim_{h \to 0} W(\mathbf{x} - \mathbf{x}', h) = \delta(\mathbf{x} - \mathbf{x}'). \tag{2.20}$$

Então, a **Integral Interpolante** $\prec A(\mathbf{x}) \succ$ de uma função $A(\mathbf{x})$ é definida como

$$\prec A(\mathbf{x}) \succ = \int_{\Omega} A(\mathbf{x}')W(\mathbf{x} - \mathbf{x}')d\mathbf{x}'. \tag{2.21}$$

Prosseguindo, ao se discretizar 2.21 obtém-se o **Somatório Interpolante** $\langle A(\mathbf{x}) \rangle$ da função $A(\mathbf{x})$ dada por

$$\langle A(\mathbf{x}) \rangle = \sum_{b=1}^{N} A(\mathbf{x}_b)W(\mathbf{x} - \mathbf{x}_b, h)\Delta\mathbf{V}_b. \tag{2.22}$$

No SPH o domínio é discretizado por um número finito de partículas. Cada partícula b ocupa um volume no espaço $\Delta\mathbf{V_b}$, possui massa m_b e densidade de massa ρ_b, portanto pode-se substituir $\Delta\mathbf{V_b}$ por $\frac{m_b}{\rho_b}$ na equação 2.22 e obter finalmente a aproximação de $A(\mathbf{x})$ usando o KI

$$\langle A(\mathbf{x}) \rangle = \sum_{b=1}^{N} \frac{m_b}{\rho_b} A(\mathbf{x}_b)W(\mathbf{x} - \mathbf{x}_b, h). \tag{2.23}$$

Pode-se mostrar que a aproximação obtida em 2.21 é de segunda ordem, ou seja,

$$A(\mathbf{x}) = \prec A(\mathbf{x}) \succ + \mathcal{O}(h^2), \tag{2.24}$$

onde o parâmetro h é o comprimento característico de W [39]. Este parâmetro é semelhante ao espaçamento das grades nos métodos Eulerianos, sendo conhecido como comprimento (ou raio) de suavização (*smoothing length*) no método SPH.

2.3.2 Características das funções Núcleo (*Kernel*)

As funções Núcleo devem preencher alguns requerimentos, por exemplo, acurácia, suavidade e eficiência computacional [37]. Uma das primeiras funções usadas por [19] foi a exponencial, em uma dimensão por exemplo,

$$W(x,h) = \frac{1}{h\sqrt{\pi}} \mathrm{e}^{-\frac{x^2}{h^2}}. \tag{2.25}$$

A exponencial é interessante pois tem infinitas derivadas, aumentando a acurácia do método [19, 52]. Todavia, a exponencial não possui suporte compacto, o que motivou o uso de splines de ordem 3 e 5. Também vale notar que splines são, a priori, computacionalmente mais eficientes [37].

Ao longo da história do método SPH foram reunidas propriedades que as funções Kernel devem possuir por vários motivos. Abaixo seguem algumas dessas propriedades e suas motivações.

- **Suavidade:** O KI é facilmente estendido para o cálculo das derivadas de uma função qualquer, todavia esta função deve ter o número necessário de derivadas para tais aproximações. Mais detalhes na seção 2.3.4.

- **Suporte compacto:** O somatório padrão do KI, como visto na definição 2.23, envolve todas as partículas, porém limitando-se o suporte da função Núcleo, isto é, $W(\mathbf{x}, h) = 0$ se $\|\mathbf{x}\| \geq kh$ onde k é uma constante definida de acordo com a ordem da função Núcleo, reduz-se drasticamente o número de partículas que entram neste somatório. A vantagem desta aproximação é reduzir o custo

computacional, cálculo do KI, de $\mathcal{O}(N^2)$ para $\mathcal{O}(nN)$, onde n é o número médio de partículas vizinhas a um ponto que se deseja calcular o valor de uma função.

- **Positividade:** A função Núcleo deve ser positiva, isto é, $W(\mathbf{x}, h) \geq 0 \ \ \forall \mathbf{x} \in \Omega$, para se garantir que significados não físicos não ocorram. Um exemplo prático é usar o KI para o cálculo da função densidade de massa. Caso a função Núcleo seja não positiva em alguma região $\Omega' \in \Omega$, então haverá contribuição de densidade de massa negativa no somatório do KI.

- **Decaimento:** Partículas mais próximas de um ponto devem ter uma influência maior no cálculo do valor de uma função deste ponto, usando o KI, do que partículas mais distantes. Pode-se garantir este comportamento escolhendo funções Núcleo que sejam monotonicamente decrescente.

- **Simetria:** A função Núcleo deve ser par, isto é, $W(\mathbf{x}, h) = W(|\mathbf{x}|, h)$, para garantir dois fatos: i) Partículas equidistantes de um ponto influenciam da mesma maneira no cálculo do valor de uma função naquele ponto. ii) A aproximação de, pelo menos, segunda ordem em h para a Representação Integral de uma função. Ver seção 2.3.1.

- **Convergência:** Se o $\lim_{h\to 0} W(\mathbf{x} - \mathbf{x}', h) = \delta(\mathbf{x} - \mathbf{x}')$ então $\langle A(\mathbf{x})\rangle \to A(\mathbf{x})$ quando $h \to 0$ [19].

- **Condição unitária:** A condição $\int_\Omega W(\mathbf{x} - \mathbf{x}', h) = 1$ garante a consistência de ondem zero (C^0) da Representação Integral de uma função.

2.3.3 Exemplos de funções Núcleo

Várias funções Núcleo foram desenvolvidas durante a existência do método SPH. Esta seção exibe algumas delas. Para simplificar a escolha da função Núcleo, foi introduzida na literatura do SPH a forma $W(\mathbf{x}, h) = \frac{\sigma_d}{h^d} f(\frac{\|\mathbf{x}\|}{h})$, onde d é a dimensão

(1,2 ou 3) e σ_d é a constante de normalização da função Núcleo. Esta dissertação também faz uso de tal forma. Abaixo os exemplos mencionados.

- Guassiana: usado em [19], esta função tem a vantagem de possuir infinitas derivadas, porém não há suporte compacto. Logo, não é computacionalmente eficiente.

$$\begin{aligned} f(x) &= \ e^{-x^2}, \quad x \geq 0 \\ \sigma &= (\sigma_1, \sigma_2, \sigma_3) = \left(\tfrac{1}{\sqrt{\pi}}, \tfrac{1}{\pi}, \tfrac{1}{(\pi\sqrt{\pi})}\right) \\ k &= +\infty \end{aligned} \tag{2.26}$$

- Forma de Sino (*Bell-Shaped*): usado em [29], esta função tem a forma semelhante a Gaussiana, porém possui suporte compacto.

$$\begin{aligned} f(x) &= \begin{cases} (1+3x)(1-x)^3, & 0.0 \leq x < 1.0 \\ 0, & 1.0 \leq x \end{cases} \\ \sigma &= \left(\tfrac{5}{4}, \tfrac{5}{\pi}, \tfrac{105}{16\pi}\right) \\ k &= 1.0 \end{aligned} \tag{2.27}$$

- Spline de Terceira Ordem: usado em [37].

$$\begin{aligned} f(x) &= \begin{cases} 1 - \frac{3}{2}x^2 + \frac{3}{4}x^3, & 0.0 \leq x < 1.0 \\ \frac{1}{4}(2-x)^3, & 1.0 \leq x < 2.0 \\ 0, & 2.0 \leq x \end{cases} \\ \sigma &= \left(\tfrac{2}{3}, \tfrac{10}{7\pi}, \tfrac{1}{\pi}\right) \\ k &= 2.0 \end{aligned} \tag{2.28}$$

- Spline de Quarta Ordem: usado em [42]

$$\begin{aligned} f(x) &= \begin{cases} (2.5-x)^4 - 5(1.5-x)^4 + 10(0.5-x)^4, & 0.0 \leq x < 0.5 \\ (2.5-x)^4 - 5(1.5-x)^4, & 0.5 \leq x < 1.5 \\ (2.5-x)^4, & 1.5 \leq x < 2.5 \\ 0, & 2.5 \leq x \end{cases} \\ \sigma &= \left(\tfrac{1}{24}, \tfrac{96}{1199\pi}, \tfrac{1}{20\pi}\right) \\ k &= 2.5 \end{aligned} \tag{2.29}$$

- Spline de Quinta Ordem: usado em [42]

$$f(x) = \begin{cases} (3-x)^5 - 6(2-x)^5 + 15(1-x)^5, & 0.0 \leq x < 1.0 \\ (3-x)^5 - 6(2-x)^5, & 1.0 \leq x < 2.0 \\ (3-x)^5, & 2.0 \leq x < 3.0 \\ 0, & 3.0 \leq x \end{cases}$$
$$\sigma = \left(\frac{1}{120}, \frac{7}{478\pi}, \frac{1}{120\pi}\right) \tag{2.30}$$
$$k = 3.0$$

- Polinômio de Wendland: usado em [31]

$$f(x) = \begin{cases} (2-x)^4(1+2x), & 0.0 \leq x < 2.0 \\ 0, & 2.0 \leq x \end{cases}$$
$$\sigma = \left(\frac{3}{64}, \frac{7}{64\pi}, \frac{21}{256\pi}\right) \tag{2.31}$$
$$k = 2.0$$

Para efeito de comparação entre as funções núcleo, a figura 2.1 apresenta o gráfico (1D) de cada função núcleo apresentada acima e suas respectivas derivadas, primeira e segunda. Para facilitar o entendimento, considere os núcleos como funções apenas de $\|\mathbf{x}\|$, normalizados e $h = 1$, *i.e.*, $W(q), q \geq 0$.

2.3.4 Cálculo do gradiente e do divergente

Seja uma função $A(\mathbf{x})$ qualquer e $\nabla_{\mathbf{x}} \equiv \frac{\partial}{\partial \mathbf{x}}$, o cálculo do gradiente $\nabla A(\mathbf{x})$ usando o KI é dado por

$$\langle \nabla A(\mathbf{x}) \rangle = \sum_{b=1}^{N} \frac{m_b}{\rho_b} A(\mathbf{x}_b) \nabla_{\mathbf{x}} W(\mathbf{x} - \mathbf{x}_b, h). \tag{2.32}$$

Para deduzir tal expressão, começa-se com a identidade trivial

$$\nabla A(\mathbf{x}) = \int_{\Omega} \left[\nabla_{\mathbf{x}'} A(\mathbf{x}')\right] \delta(\mathbf{x} - \mathbf{x}') d\mathbf{x}', \tag{2.33}$$

e em seguida substitui-se a função $\delta(\mathbf{x} - \mathbf{x}')$ por $W(\mathbf{x} - \mathbf{x}')$ em (2.33), de modo semelhante àquele utilizado para obter a equação (2.21), para então executar os

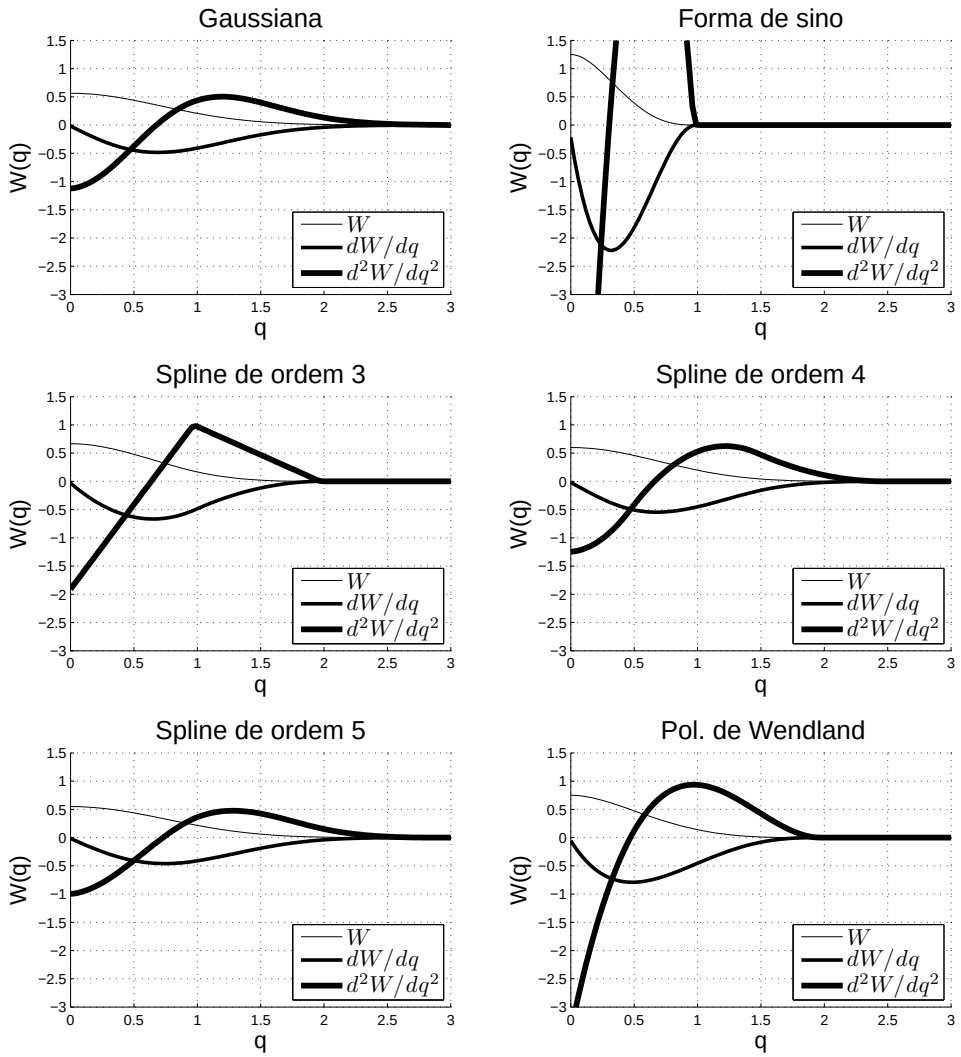

Figura 2.1: Gráfico das funções núcleo e suas respectivas derivadas. Para facilitar o entendimento, considere os núcleos como funções apenas de $\|\mathbf{x}\|$, normalizados e $h = 1$, *i.e.*, $W(q), q \geq 0$.

seguintes passos

$$\begin{aligned}
\nabla A(\mathbf{x}) &= \int_\Omega \left[\nabla_{\mathbf{x}'} A(\mathbf{x}')\right] W(\mathbf{x}-\mathbf{x}')d\mathbf{x}' + \mathcal{O}(h^2) \\
&= \int_\Omega \left\{\nabla_{\mathbf{x}'}\left[A(\mathbf{x}')W(\mathbf{x}-\mathbf{x}')\right] - A(\mathbf{x}')\nabla_{\mathbf{x}'}W(\mathbf{x}-\mathbf{x}')\right\} d\mathbf{x}' + \mathcal{O}(h^2) \\
&= \underbrace{\int_\Omega \nabla_{\mathbf{x}'}\left[A(\mathbf{x}')W(\mathbf{x}-\mathbf{x}')\right] d\mathbf{x}'}_{I_1} - \int_\Omega A(\mathbf{x}')\nabla_{\mathbf{x}'}W(\mathbf{x}-\mathbf{x}')d\mathbf{x}' + \mathcal{O}(h^2) \\
&= \underbrace{\int_{\partial\Omega} A(\mathbf{x}')W(\mathbf{x}-\mathbf{x}')d\mathbf{S}}_{I_2} - \int_\Omega A(\mathbf{x}')\nabla_{\mathbf{x}'}W(\mathbf{x}-\mathbf{x}')d\mathbf{x}' + \mathcal{O}(h^2) \\
&= -\int_\Omega A(\mathbf{x}')\nabla_{\mathbf{x}'}W(\mathbf{x}-\mathbf{x}')d\mathbf{x}' + \mathcal{O}(h^2) \\
&\overset{(1)}{=} \underbrace{\int_\Omega A(\mathbf{x}')\nabla_{\mathbf{x}}W(\mathbf{x}-\mathbf{x}')d\mathbf{x}'}_{I_3} + \mathcal{O}(h^2) \\
&= \prec \nabla A(\mathbf{x}) \succ + \mathcal{O}(h^2),
\end{aligned}$$

onde foi usado o Teorema de Gauss para substituir a integral I_1 por I_2 e, como a função W normalmente tem suporte compacto, ou seja, vale zero na fronteira, I_2 zera. Por último, a igualdade (1) foi obtida pois, sendo

$$\nabla W(\mathbf{u}) \equiv \frac{\mathbf{u}}{\|\mathbf{u}\|}\frac{\partial W}{\partial\|\mathbf{u}\|} \tag{2.34}$$

e

$$\nabla_{\mathbf{x}} W(\mathbf{x}-\mathbf{x}') \equiv \frac{\partial W}{\partial \mathbf{x}} = \frac{\partial W}{\partial \mathbf{u}}\frac{\partial \mathbf{u}}{\partial \mathbf{x}} = \nabla W(\mathbf{u})(-1), \tag{2.35}$$

$$\nabla_{\mathbf{x}'} W(\mathbf{x}-\mathbf{x}') \equiv \frac{\partial W}{\partial \mathbf{x}'} = \frac{\partial W}{\partial \mathbf{u}}\frac{\partial \mathbf{u}}{\partial \mathbf{x}'} = \nabla W(\mathbf{u})(1), \tag{2.36}$$

portanto,

$$\nabla_{\mathbf{x}} W(\mathbf{x}-\mathbf{x}') = -\nabla_{\mathbf{x}'} W(\mathbf{x}-\mathbf{x}') = -\nabla W(\mathbf{u}). \tag{2.37}$$

A equação (2.32) é finalmente obtida fazendo-se a discretização da integral I_3 seguindos passos análogos àqueles feitos para derivar (2.23) a partir de (2.21).

A dedução do divergente de uma função vetorial $\mathbf{B}(\mathbf{x})$ segue o mesmo molde da dedução do gradiente de uma função escalar visto acima. Portanto, alguns detalhes serão omitidos.

A expressão a ser deduzida é:

$$\langle \nabla \cdot \mathbf{B}(\mathbf{x}) \rangle = \sum_{b=1}^{N} \frac{m_b}{\rho_b} \mathbf{B}(\mathbf{x}_b) \cdot \nabla_{\mathbf{x}} W(\mathbf{x} - \mathbf{x}_b, h). \tag{2.38}$$

Começando com uma identidade trivial, a dedução de (2.38) é dada pelos seguintes passos:

$$\begin{aligned}
\nabla \cdot \mathbf{B}(\mathbf{x}) &= \int_{\Omega} \left[\nabla_{\mathbf{x}'} \cdot \mathbf{B}(\mathbf{x}')\right] \delta(\mathbf{x} - \mathbf{x}') d\mathbf{x}', \\
&= \int_{\Omega} \left[\nabla_{\mathbf{x}'} \cdot \mathbf{B}(\mathbf{x}')\right] W(\mathbf{x} - \mathbf{x}') d\mathbf{x}' + \mathcal{O}(h^2) \\
&= \int_{\Omega} \left\{\nabla_{\mathbf{x}'} \cdot \left[\mathbf{B}(\mathbf{x}') W(\mathbf{x} - \mathbf{x}')\right] - \mathbf{B}(\mathbf{x}') \cdot \nabla_{\mathbf{x}'} W(\mathbf{x} - \mathbf{x}')\right\} d\mathbf{x}' + \mathcal{O}(h^2) \\
&= \int_{\Omega} \nabla_{\mathbf{x}'} \cdot \left[\mathbf{B}(\mathbf{x}') W(\mathbf{x} - \mathbf{x}')\right] d\mathbf{x}' - \int_{\Omega} \mathbf{B}(\mathbf{x}') \cdot \nabla_{\mathbf{x}'} W(\mathbf{x} - \mathbf{x}') d\mathbf{x}' + \mathcal{O}(h^2) \\
&= \int_{\partial\Omega} \mathbf{B}(\mathbf{x}') W(\mathbf{x} - \mathbf{x}') d\mathbf{S} - \int_{\Omega} \mathbf{B}(\mathbf{x}') \cdot \nabla_{\mathbf{x}'} W(\mathbf{x} - \mathbf{x}') d\mathbf{x}' + \mathcal{O}(h^2) \\
&= -\int_{\Omega} \mathbf{B}(\mathbf{x}') \cdot \nabla_{\mathbf{x}'} W(\mathbf{x} - \mathbf{x}') d\mathbf{x}' + \mathcal{O}(h^2) \\
&= \int_{\Omega} \mathbf{B}(\mathbf{x}') \cdot \nabla_{\mathbf{x}} W(\mathbf{x} - \mathbf{x}') d\mathbf{x}' + \mathcal{O}(h^2) \\
&= \prec \nabla \cdot \mathbf{B}(\mathbf{x}) \succ + \mathcal{O}(h^2),
\end{aligned}$$

A expressão para calcular o gradiente de uma função, dada por (2.32), possui variações, pois há mais de uma maneira de expressar o gradiente de uma função na forma contínua, *i.e.*,

$$\nabla A = \frac{1}{\rho} \left[\nabla (\rho A) - A \nabla \rho\right] \tag{2.39}$$

ou

$$\nabla A = \rho \left[\nabla \left(\frac{A}{\rho}\right) + \frac{A}{\rho^2} \nabla \rho\right], \tag{2.40}$$

Estas expressões, quando utilizadas, levam a diferentes discretizações no SPH. Por exemplo, combinando (2.32) e (2.39) resulta em

$$\langle \nabla A(\mathbf{x}) \rangle = \frac{1}{\rho} \sum_{b=1}^{N} m_b \left(A(\mathbf{x}_b) - A(\mathbf{x}) \right) \nabla_{\mathbf{x}} W(\mathbf{x} - \mathbf{x}_b, h), \tag{2.41}$$

possuindo, segundo [39], a característica de ter maior acurácia em relação à formulação original dada por (2.32), porém é anti-simétrica. Para ver esta última característica basta tomar como exemplo duas partículas vizinhas e calcular a influência mútua entre elas.

Caso (2.32) for combinado com (2.40), então

$$\langle \nabla A(\mathbf{x}) \rangle = \rho \sum_{b=1}^{N} m_b \left(\frac{A(\mathbf{x})}{\rho^2} + \frac{A(\mathbf{x}_b)}{\rho_b^2} \right) \nabla_{\mathbf{x}} W(\mathbf{x} - \mathbf{x}_b, h). \tag{2.42}$$

resulta em uma formulação simétrica, conservando-se assim os momentos linear e angular, também segundo [39]. Esta característica torna o uso desta formulação, para o cálculo do gradiente da pressão no SPH, bastante comum. Em [11] é feita uma comparação entre as três diferentes formulações do gradiente, dadas pelas equações (2.32), (2.41) e (2.42).

De maneira semelhante, pode-se derivar outras formulações para o divergente de uma função, dada por (2.38). Por exemplo,

$$\langle \nabla \cdot \mathbf{B}(\mathbf{x}) \rangle = \frac{1}{\rho} \sum_{b=1}^{N} m_b \left(\mathbf{B}(\mathbf{x}_b) - \mathbf{B}(\mathbf{x}) \right) \cdot \nabla_{\mathbf{x}} W(\mathbf{x} - \mathbf{x}_b, h), \tag{2.43}$$

e

$$\langle \nabla \cdot \mathbf{B}(\mathbf{x}) \rangle = \rho \sum_{b=1}^{N} m_b \left(\frac{\mathbf{B}(\mathbf{x})}{\rho^2} + \frac{\mathbf{B}(\mathbf{x}_b)}{\rho_b^2} \right) \cdot \nabla_{\mathbf{x}} W(\mathbf{x} - \mathbf{x}_b, h) \tag{2.44}$$

podem ser derivadas. Assim como o gradiente, os divergentes na forma (2.43) e (2.44) são anti-simétrico e simétrico, respectivamente. Além destas, outras formas, tanto do gradiente quanto do divergente, são apresentadas em [39].

2.4 Discussão e Conclusões

A análise do MA-SPH feita em [22] permitiu entender os TSE comumente empregados no SPH. Para fazer a análise, a autora de [22] separou os TSE's em três categorias: i) tratamentos de acurácia; ii) tratamentos de consistência e iii) tratamentos de estabilidade. Essa categorização se deve ao fato de, normalmente, cada tratamento tentar resolver uma única questão do SPH.

2.4.1 Tratamentos de acurácia

Segundo a análise de [22] o MA-SPH permite soluções espúrias de frequência altamente oscilatória (FAO) devido à hipótese da compressibilidade fraca, além da solução incompressível. Em particular, foi mostrado que essas soluções dominam a dinâmica da solução, isto é pressão e aceleração, porém quantidades temporalmente integráveis no tempo, tais como a posição de cada partícula na superfície livre, não exibiram significante FAO. Comparando dados experimentais de elevação de superfície em fluxos altamente não lineares e complexos, pôde-se validar a hipótese de compressibilidade fraca. Contudo, a dinâmica correspondente exibiu erros da ordem de até 100%.

Alguns autores, percebendo erro no valor da pressão nos pontos de interesse das simulações, criaram, por exemplo, os seguintes tratamentos para tentar remover as FAO's da pressão:

- Filtragem temporal da pressão: para tentar corrigir a pressão, este tratamento se baseia no pós-processamento, *i.e.*, aplica-se uma frequência de corte usando uma filtragem de Fourier [21]. Os resultados obtidos com esse tratamento vão de acordo com as dados experimentais.

- Filtragem espacial da pressão: também é um pós-processamento, onde se usa o KI para obter uma pressão média [47]. Os resultados desse tratamento não se mostraram efetivos.

2.4.2 Tratamentos de consistência

Antes do trabalho [22] havia uma discrepância entre resultados analíticos e numéricos com relação à convergência do KI. Algumas análises mostravam que o KI divergia, mas outras mostravam convergência numérica. Então, vários TSE's foram desenvolvidos para contornar tal situação. Todavia, é provada em [22] a consistência do KI no contexto do SPH, sendo a idéia geral de divergência do KI próximo à fronteira expressa em termos de uma dinâmica espúria da condição de superfície livre. Essa dinâmica espúria é dependente da inclinação da fronteira livre.

Também ficou provada, em [22], a advecção suavizada das partículas inicialmente localizadas em uma grade uniforme e ilimitada, que resulta no movimento das partículas no SPH.

Como o KI é consistente no contexto do SPH, os tratamentos criados pelos seus autores podem ser vistos como: i)formas alternativas de computar as derivadas espaciais ou ii)formas discretas diferentes da equação de movimento.

No primeiro caso, o *Moving Least Square* (MLS) [24], mais computacionalmente custoso que o KI, é representante desta categoria. Outro representante é o método Müller [45]. Ambos utilizam matrizes para o cálculo das derivadas parciais e dependendo da posição das partículas as matrizes podem se tornar singulares. Portanto, tanto um como o outro, não são robustos para fluxos altamente não lineares.

No segundo caso, as diferentes formas de equações de movimento podem ser divididas em dois grupos: i) aqueles tratamentos relacionados à equação do momento [48] e ii) os relacionados à conservação da massa. Em geral, as formulações relacionadas à equação do momento focam na conservação do momento, ignorando a presença das fronteiras. Formulações relacionadas à conservação de massa [26] focam na inconsistência do KI próxima a fronteira livre. Os dois grupos de tratamentos mencionados melhoram a consistência do KI próximo à fronteira livre.

2.4.3 Tratamentos de estabilidade

Ficou provado em [22] a instabilidade inerente do MA-SPH. São três as causas desta instabilidade e cada uma é descrita separadamente a seguir. Logo após são exibidos alguns dos TSE's aplicados para tentar contornar o problema da instabilidade do SPH.

- Instabilidade da compressibilidade fraca: as maiores taxas de instabilidade são devidas à hipótese de compressibilidade fraca na presença da superfície livre [54] e eventualmente dominam as simulações de longa duração.

- Componentes oscilatórios: são componentes instáveis na presença de gradiente de densidade base não zero. Na literatura este tipo de instabilidade é conhecido como instabilidade tensorial e erroneamente associado à discretização espacial [41, 42, 1, 3, 57].

- Instabilidade temporal: é a instabilidade proveniente do esquema de integração temporal adotado e depende, entre outras quantidades, da condição de Courant $\mu_c = \frac{\delta t\, c}{h}$, onde δt é o passo de tempo, c é a velocidade artificial do som e h o comprimento de suavização.

As análises anteriores ao trabalho de [22] estavam limitadas ao estudo da instabilidade tensorial, que erroneamente foi associada ao KI. Contudo, o problema reside no fato de que a maior fonte de instabilidade está na existência de gradiente de densidade base não zero. Abaixo, alguns dos TSE's analisados:

- Velocidade Artificial [40]: adiciona-se um termo viscoso artificial no lado direito da equação do momento. A análise do tratamento indicou que o mesmo parece não levar a uma regularização geral do método.

- XSPH [38]: neste tratamento, a posição de cada partícula é atualizada não apenas usando a velocidade da mesma, mas também uma velocidade artificial. O tratamento vai de acordo com as observações numéricas [41, 10], embora pareça não levar a uma estabilização geral do método.

- Remoção inicial (*Initial Dumping*) [40]: baseia-se na remoção de energia cinética do fluido antes da inicialização do fluxo. A análise deste método indicou que o método pode ser regularizado numericamente através da escolha apropriada de um *dumping* numérico.

- Tratamento de instabilidade tensorial [41]: da mesma maneira que o tratamento da Velocidade Artificial, parece não levar a uma regularização geral do método.

- Esquema de reinicialização da densidade suavizada [45]: este esquema dissipa a densidade, mas não remove gradientes de densidade, *i.e.*, não possui efeitos significativos na física simulada. A conclusão é que a dissipação introduzida é robusta e suave.

3 FUNDAMENTOS DO MSPH, UM MÉTODO SPH BASEADO NO KI.

3.1 Introdução

O método mSPH foi criado em [22] a partir do MA-SPH, retendo a hipótese de compressibilidade fraca e o cálculo das derivadas espaciais usando o KI. Além disso, remove todos os TSE's existentes no SPH. A base para a construção deste algoritmo está no resultado das análises do método MA-SPH e dos tratamentos semi-empíricos existentes, ambas as análises feitas em [22].

As principais características do método são:

- É de primeira ordem.
- Retém a simplicidade e robustez do método SPH.
- Permite passos de tempo maiores que o SPH.
- Remove os parâmetros ajustáveis do SPH.
- Por último e mais importante, segundo a autora, é dissipativo, ou seja, é estável com propriedades de convergência conhecidas, conforme a autora.

E, para conservar a robustez do MA-SPH, no mSPH permite-se a geração de modos acústicos espúrios para posteriormente removê-los, impondo-se uma dissipação numérica conhecida de forma apropriada.

A análise do MA-SPH para fluxos com fronteira livre, feita em [22], elucidou três questões-chave:

1. A hipótese de Compressibilidade Fraca permite o desenvolvimento de modos acústicos espúrios nas simulações. O desenvolvimento destes modos tem origem em:

- Implementações numéricas inconsistentes das condições iniciais e de fronteira, *e.g.*, não levar em conta a diferença de pressão hidrostática entre partículas do fluido e partículas de fronteira fixa.
- Instabilidades.

2. A incompletude do KI próximo à fronteira livre produz uma condição dinâmica espúria de fronteira livre que é função do próprio formato da fronteira livre.

3. Três tipos de instabilidades podem se desenvolver:

   - Inerente instabilidade incondicional de fluxos fracamente compressíveis em domínios semi-infinitos.
   - Instabilidade tensorial presente em fluxos com gradiente de densidade base diferente de zero.
   - Instabilidade por causa de esquemas de integração numérica que dependem da condição de Courant.

Devido à ausência de análises conclusivas do SPH antes do trabalho feito em [22], vários TSE's foram desenvolvidos, podendo ser classificados em tramentos de: i) acurácia (remoção de modos acústicos); ii) consistência (abordando principalmente a condição de fronteira livre) e iii) estabilidade. A análise dos tratamentos mostrou que eles geralmente apontam na direção correta, mas são insuficientes.

Por reter a hipótese de compressibilidade fraca, o mSPH carrega consigo um erro aceitável da ordem de $O(1/c^2)$, dado que c é a velocidade artificial do som, junto com modos acústicos linearmente dependentes em c. Os modos acústicos são predominantemente gerados por condições inconsistentes iniciais e de fronteira e também por ruido numérico. No SPH, esses modos eventualmente dominam a dinâmica devido a instabilidades numéricas.

No intuito de minimizar a geração de modos acústicos espúrios no mSPH, inconsistências entre equações governantes e condições de fronteira são removidas.

Para isto, emprega-se a equação linear do estado (2.5) no lugar de (2.4), reformula-se as equações governantes para capturar a pressão hidrostática incompressível e, por último, corrige-se as condições de fronteira para levar em conta a diferença de pressão hidrostática.

No mSPH a questão-chave relacionada à incompletude do KI próximo à fronteira livre não recebe qualquer tratamento de consistência. A dinâmica espúria causada por esta incompletude é livremente gerada e posteriormente dissipada usando-se uma suavização periódica das variáveis de campo do fluxo.

A estabilidade do mSPH é obtida através do emprego de um esquema de integração temporal de ordem mais alta junto com uma suavização periódica das variáveis de campo do fluxo. A suavização é responsável por introduzir uma dissipação conhecida.

3.2 Formulação do mSPH

3.2.1 Equações governantes

As equações governantes do mSPH, na forma contínua, são:

$$\frac{d\mathbf{u}}{dt} = -\frac{1}{\rho_f}\nabla P + \mathbf{g} \tag{3.1}$$

$$\frac{d\rho}{dt} = -\rho\nabla\cdot\mathbf{u} \tag{3.2}$$

$$dP = c^2 d\rho. \tag{3.3}$$

Portanto, a equação incompressível de Euler para o momento (3.1), a conservação da massa para a densidade (3.2) e a equação de estado (3.3) linear para o cálculo da pressão P governam o fluido na formulação mSPH.

3.2.2 Discretização espacial

A discretização do mSPH é análoga à do MA-SPH, *i.e.*, o fluxo é discretizado em $N \gg 1$ partículas com espaçamento inicial $O(h)$, cada uma carregando sua própria massa constante m_a, e propriedades de campo tais como densidade $\rho_a(t)$,

pressão $P_a(t)$ e velocidade $\mathbf{u}_a(t)$. As equações governantes para o fluxo discretizado são:

$$\frac{d\mathbf{x}_a}{dt} = \mathbf{u}_a \tag{3.4}$$

$$\frac{d\mathbf{u}_a}{dt} = -\frac{1}{\rho_f}\sum_{b=1}^{N}(P_a + P_b)\frac{m_b}{\rho_b}\nabla W_{ab} + \mathbf{g} \tag{3.5}$$

$$\frac{d\rho_a}{dt} = -\rho_a \sum_{b=1}^{N}\frac{m_b}{\rho_b}(\mathbf{u}_a - \mathbf{u}_b)\cdot\nabla W_{ab}, \tag{3.6}$$

$$P_a = c^2(\rho_f - \rho_a). \tag{3.7}$$

A inclusão dos termos P_a e $\mathbf{u}_a$ nas equações da conservação do momento e da massa, respectivamente, afeta principalmente a implementação da condição dinâmica da fronteira livre. Deste modo, há uma atenuação dos modos acústicos gerados pela inconsistência devido à incompletude do KI próximo à fronteira livre.

3.2.3 Condições de fronteira

As condições de fronteira do método mSPH são as mesmas do MA-SPH, ou seja:

1. $\mathbf{u}\cdot\hat{n} = 0$, onde $\hat{n}$ é a normal da fronteira não permeável.
2. $P(x, y = \eta, t) = 0$, pressão zero na superfície livre.
3. $\dot{y}_{fs} = \frac{d\eta(x,t)}{dt}$, continuidade cinemática da superfície livre.

E também da mesma forma que no MA-SPH, as condições de fronteira acima são modeladas da seguinte forma, respectivamente:

1. Uso de partículas fantasmas, espelhadas com relação à fronteira impermeável [10].
2. Condição não é imposta, pois o KI é incompleto na superfície livre. A estratégia é usar o KI normalmante, gerando erros numéricos, e posteriormente suavizar tais erros. [22].

3. Satisfeita automaticamente, pois em todos os métodos lagrangianos tem-se $\frac{dy_a}{dt} = \frac{d\eta}{dt}$ para $a \in \partial_{fs}$.

A condição de fluxo zero na fronteira impermeável deve ser satisfeita levando-se em conta a diferença hidrostática de pressão, *i.e.*, o fluido e as partículas fantasmas devem ter a mesma pressão dinâmica para que $\frac{d\mathbf{u}_a}{dt} \cdot \hat{n}_{ib} \to 0$ para $a \to \partial_{ib}$, em que $\hat{n}_{ib}$ denota a normal da superfície impermeável.

Para exemplificar o uso de partículas fantasmas na condição de fluxo zero em fronteiras impermeáveis, são considerados apenas fronteiras planas. Seja a uma partícula de fluido localizada a uma distância dy da fronteira impermeável ∂_b dada pela função $y = -H$ com $\mathbf{x}_a = \{x_a, y_a\}$, $\mathbf{u}_a = \{u_a, v_a\}$ e ρ_a, então se $dy \leq \alpha h$, e $\alpha \sim O(5)$, a partícula é espelhada perpendicularmente à fronteira e a partícula fantasma a^* é criada da seguinte forma:

$$x_a^* = x_a, y_a^* = -H - dy, \tag{3.8}$$

$$u_a^* = u_a, v_a^* = -v_a, \tag{3.9}$$

$$\rho_a^* = \rho_a + \rho_f \frac{g}{c^2}(y_a - y_a^*), \tag{3.10}$$

O termo adicional na equação (3.10) é para estabelecer a diferença de pressão hidrostática entre as partículas.

3.2.4 Condições iniciais

As condições iniciais do fluxo devem ser consistentes com as equações governantes e condições de fronteira. Caso o fluido tenha uma componente hidrostática inicial, então esta deve ser levada em conta no cálculo da densidade inicial das partículas assim como a massa utilizada no KI.

Para exemplificar, considere uma simulação de quebra de represa (*dam-break* [58]) em que o fluxo é iniciado como hidrostático. No instante $t = 0$ as partículas estão em uma grade bidimensional regular com espaçamento h entre os nós. Seja H a altura de referência da pressão hidrostática, as partículas são inicializadas da

seguinte maneira:

$$\mathbf{x}_a = [a_x h, a_y h], \tag{3.11}$$

$$\mathbf{u}_a = 0, \tag{3.12}$$

$$\rho_a = \rho_f(1 + \frac{g}{c^2}(H - a_y h)),\ e \tag{3.13}$$

$$m_a = \rho_a h^2. \tag{3.14}$$

Pode-se notar a consistência entre estas condições iniciais e as equações governantes do mSPH dada por (3.5), (3.6) e (3.7). É importante destacar que condições iniciais inconsistentes irão gerar modos acústicos espúrios.

3.2.5 Integração temporal

O esquema de integração temporal das equações governantes do mSPH adotado é o Runge-Kutta de quarta ordem. Seja o passo de tempo δt e $q^n \equiv q(t = n\delta t) = [\mathbf{x}^n, \mathbf{u}^n, \rho^n]$ a variável vetorial do fluxo, a integração é dada por:

$$\begin{cases} q^{1/4} & = q^n + \dot{q}^n \frac{\delta t}{2} \\ q^{1/2} & = q^n + \dot{q}^{1/4} \frac{\delta t}{2} \\ q^{3/4} & = q^n + \dot{q}^{1/2} \frac{\delta t}{2} \\ q^{n+1} & = q^n + (\dot{q}^n + 2\dot{q}^{1/4} + 2\dot{q}^{1/2} + \dot{q}^{3/4}) \frac{\delta t}{6} \end{cases} \tag{3.15}$$

onde $\dot{q}^m$ denota a taxa de variação temporal das variáveis de campo do fluxo em um tempo $t = m\delta t$. O passo de tempo é dado por $\delta t = \mu_c \frac{h}{c}$, através da condição de Courant μ_c. Valores possíveis de μ_c foram encontrados em [22] usando-se a análise de estabilidade de von Neumann (Von Neumann e Riclitmeyer, 1950) do método MA-SPH.

3.2.6 Regularização do mSPH

O esquema mSPH descrito é inerentemente instável tal como o SPH, portanto alguma forma de regularização deve ser aplicada. A análise dos TSE's indicou que os tratamentos mais efetivos usavam alguma forma de dissipação. Então, o mSPH segue este caminho, tendo em mente que a dissipação deve ser aplicada em todo

o fluido. Na prática, o mSPH é regularizado através de uma suavização periódica usando funções de Shepard. A cada $n = f(\mu)$ passos de tempo [22], o campo é atualizado baseado em:

$$\mathbf{u}_a^s = \frac{\sum_b \frac{m_b}{\rho_b} \mathbf{u}_b W_{ab}}{\sum_b \frac{m_b}{\rho_b} W_{ab}} \tag{3.16}$$

$$\rho_a^s = \frac{\sum_b m_b W_{ab}}{\sum_b \frac{m_b}{\rho_b} W_{ab}} \tag{3.17}$$

4 UMA IMPLEMENTAÇÃO DO MSPH USANDO GPGPU

Para contextualizar a implementação do mSPH usando GPGPU, este capítulo se inicia discorrendo sobre o surgimento do hardware gráfico programável, a evolução desde hardware até os dias atuais, compara a CPU com a GPU e, por último, a plataforma de software OpenCL para programação paralela em GPU utilizada nesta dissertação.

Posteriormente, a seção 4.2 detalha cada primitiva usada para compor esta implementação. Nesta mesma seção, uma atenção maior é dada ao algoritmo de detecção de colisão, devido ao seu grau de complexidade. A seção seguinte 4.3 fica encarregada de explicar a composição da implementação computacional do mSPH em GPU, proposta por esta dissertação.

4.1 Introdução

Renderização, no contexto da computação gráfica, pode ser vista como um processo com a seguinte finalidade: geração de imagens bidimensionais a partir de representações tridimensionais de ambientes virtuais. Estes ambientes são compostos por objetos que podem ser descritos usando-se malhas de polígonos planares, *e.g.*, triângulos. Por não ser único o processo de renderização, pode-se encontrar na literatura sobre o assunto inúmeras descrições [56], cada uma podendo ser composta por uma sequência de passos. A esta sequência de passos dá-se o nome de *pipeline* gráfico ou somente *pipeline*. Por propósitos de simplificação, esta dissertação foca apenas na renderização por rasterização.

Com o intuito de facilitar o entendimento da origem e evolução dos sistemas gráficos de geração de imagem em tempo real, um *pipeline* simplificado é usado nesta dissertação, sendo este descrito por apenas dois estágios em sequência: i) processamento de geometrias; e ii) processamento de fragmentos. O estágio de

processamento de geometrias opera sobre os vértices que compõem a malha poligonal de cada objeto virtual. Já o estágio de processamento de fragmentos recebe as primitivas transformadas na etapa anterior, gera candidatos a píxels através da rasterização e processa-os para em seguida serem gravados no *framebuffer* (memória usada pelo monitor para exibir as imagens renderizadas).

Mais detalhadamente, o estágio de processamento de fragmentos pode compreender as seguintes tarefas: i) identificar todos os píxels que se projetam no interior de um polígono; ii) determinar os píxeis visíveis de cada polígono; e iii) pintar cada píxel deste [17].

A GPU pode ser vista como uma instância de sistemas gráficos que tem como objetivo permitir a geração de imagens concebidas através de renderização. Grosso modo, estes sistemas gráficos precisam preencher o *framebuffer* com píxels, que juntos formam uma imagem. Estas imagens devem ser geradas em tempo real possibilitando interações entre usuário e ambiente virtual.

A tarefa de renderizar imagens realísticas de objetos virtuais é computacionalmente exigente. Os hardwares de propósito geral sozinhos (no início da década de 80) não davam conta desta tarefa ser executada em tempo real de maneira aceitável [15]. Diante deste cenário e durante um curso de ciência da computação de verão na Universidade da Carolina do Norte, surgiu um projeto com uma abordagem radical: acoplar processadores de píxels à memória e prover um processador para cada píxel funcionando como uma máquina SIMD. Nascia assim o Pixel-Planes (1981) [17].

Em sistemas gráficos convencionais de rasterização anteriores ao Pixel-Planes, a informação era passada ao *framebuffer* através de pares coordenadas/dado (x,y/RGB). Já a inovação proposta pelo Pixel-Planes consistia em enviar expressões lineares e processá-las simultaneamente em cada píxel. Enquanto que outros sistemas gráficos executavam cálculos de natureza gráfica mais complexos em processadores de propósito geral ou em hardwares especiais que executavam apenas um conjunto de funções gráficas [8], o Pixel-Planes pretendia ser um *engine* de propósito geral, particularmente poderoso quando operações em píxel podiam ser expressas em termos

de expressões lineares [15].

No sistema gráfico Pixel-Planes, polígonos são recebidos e processados na primeira etapa de forma serial por um processador de domínio específico, *i.e.*, produzido para este tipo de tarefa. Entretanto, o estágio de processamento de fragmentos é feito em paralelo. N idênticos *'smart' memory chips* processam em paralelo uma primitiva, *e.g.*, um triângulo, gerando N píxels por vez. Este conjunto de chips, também conhecido como *'smart' framebuffer*, ou ainda *Enhanced-Memory Chips* (EMCs), foi o foco do projeto Pixel-Planes, pois a segunda etapa do *pipeline* se mostrava computacionalmente mais onerosa que a primeira. Sendo assim, a segunda etapa se mostrava como gargalo da renderização em tempo real. O primeiro protótipo empregava um 2x2 *pixel memory array* (PMA), ou seja, 4 EMCs [17].

Mantendo basicamente a mesma arquitetura do primogênito, as versões seguintes, *i.e.*, os Pixel-Planes 2, 3 e 4 (1982, 1983 e 1987) apenas tiveram escalados o tamanho do PMA, ou seja, aumentando a quantidade de EMCs. Estes empregavam 4x64, 64x64 e 512x512 PMAs, respectivamente. Há de se destacar que uma inovação da arquitetura Pixel-Planes 4 foi a introdução do *Image Generation Controller* (IGC), uma espécie de controlador de programa, que viria a ser importante para o design da arquitetura seguinte [15].

Até a geração anterior (Pixel-Planes 4), apenas um *stream* de polígonos era processado por vez. Nesta abordagem, muitos processadores de píxel ficavam ociosos, pois a "área" ocupada por um polígono sendo processado na segunda etapa do *pipeline* era quase sempre muito menor que a "área" total do *framebuffer*. Sendo assim, muitos EMCs ficavam ociosos. Portanto, o desperdício de processamento era grande. Para contornar este problema, a nova arquitetura chamada de Pixel-Planes 5 implementou um particionamento do *framebuffer* em conjuntos disjuntos de regiões contíguas. Assim, múltiplos polígonos poderiam ser processados ao mesmo tempo. Contudo, isso não seria possível sem introduzir o poder computacional necessário, devido a essa mudança, na primeira etapa do *pipeline* de renderização. Por este motivo, introduziu-se também múltiplos processadores geométricos [18].

Portanto, o Pixel-Planes 5 introduziu mais um tipo de paralelismo, *i.e.*, o de processamento de objetos (polígonos), utilizando múltiplos processadores geométricos. Para o processamento de píxels em paralelo introduziu-se múltiplos renderizadores, e cada um deste era composto de um PMA de 128x128 e um IGC. Devido ao uso de IGC, cada processador de geometria ficou responsável por controlar um ou mais renderizadores. Com esta nova arquitetura, conseguiu-se uma taxa de desenho de 1 milhão de polígonos por segundo, cerca de 28 vezes mais do que quando comparado à geração anterior [18].

Apesar do Pixel-Planes 5 exibir um aumento de desempenho bastante significativo em relação ao Pixel-Planes 4, ele possuía algumas severas restrições para se conseguir aumento de desempenho. A principal restrição é a não escalabilidade do número de processadores (tanto geométricos como renderizadores), uma vez que estes dependiam da velocidade e da largura de banda da rede que os conectavam. Uma outra restrição acontece quando um grupo de primitivas ficam amontoadas em um conjunto pequeno de regiões de píxels acarretando um problema de balanceamento de carga [16].

Todas os projetos Pixel-Planes são baseados em algoritmos de divisão de tela, pois cada PMA fica responsável por uma região desta. O problema é que esta abordagem não escala indefinidamente. Para superar este entrave a arquitetura seguinte adotou uma nova abordagem, um algoritmo baseado em composição de imagem. Nesta abordagem, cada par processador geométrico/renderizador é responsável por gerar um quadro completo (em inglês *frame*, *i.e.,* uma imagem) a partir de um subconjunto de primitivas que lhe foi atribuído. Para a geração de uma imagem final mais uma tarefa é necessária: a de composição. Como o nome já indica, esta tarefa é responsável por compor o quadro final usando todas os quadro parciais gerados pelos renderizadores. Para isto, pode-se utilizar um algoritmo de visibilidade, *e.g.*, o *z-buffer* [36].

O nome dado à arquitetura seguinte, a qual adotou a abordagem baseada em composição de imagem descrita acima, foi PixelFlows (1992). Este conseguiu

ultrapassar em 100 vezes a taxa de renderização do Pixel-Planes 5, desenhando até 100 milhões de polígonos por segundo [36].

No mesmo ano que o PixelFlows é lançado, surge também o OpenGL, uma API gráfica, *i.e.,* uma especificação de interface de software que deve interagir com hardware gráfico. Sua especificação é independente de hardware, porém uma implementação particular deve ser usada para controlar um determinado tipo de hardware. O OpenGL é um padrão aberto que evoluiu muito ao longo dos anos e até hoje é largamente adotado.

Alguns anos depois, em 1995, surge a primeira placa de vídeo da NVIDIA, chamada de NV1. Cabe ressaltar que esta empresa é responsável pelo surgimento do nome GPU. Em 1997 surge também a primeira placa da ATI (posteriormente, esta empresa é adquirida pela AMD). Hoje em dia a AMD e a NVIDIA são as maiores fabricantes de GPU, dominando este setor no mercado mundial.

Em 1998 é exposto em [49] o primeiro sistema gráfico suportando a execução de *shaders* em tempo real. *Shader* é um programa simbiótico (sua execução depende de um programa principal), geralmente pequeno, usado para determinar a cor final de cada ponto em uma superfície. O *shader* pode ficar responsável por variações de cores ao longo da superfície e a interação de luz com a superfície [49].

Em 2000 é demonstrado que a arquitetura OpenGL pode ser tratada como um processador SIMD geral. Isto pode ser feito através da tradução de uma descrição *shader* de alto nível em passos de renderização. A idéia consistia em fazer re-circular dados através do *pipeline* gráfico [51].

Um ano depois, em 2001, apareceram as primeiras GPUs com a capacidade de executar dois tipos de *shaders* em tempo real, o de vértice e o de fragmento. A partir desta data começaram a surgir artigos relatando o uso da GPU em cálculos de propósito geral. Para isto, passa-se a considerar a GPU como um processador *stream* [53].

O modelo de programação *stream* captura a localidade não presente em modelos SIMDs ou vetoriais através do uso de *streams* e *kernels*. Um *stream* é uma

coleção de registros (itens relacionados que são tratados como uma unidade) requerendo uma computação similar. Já os *kernels* são funções aplicadas a cada elemento de um *stream*. Isto é, um processador *stream* executa um *kernel* sobre todos os elementos de uma *stream* de entrada, colocando os resultados em uma saída *stream* [5].

Na época dos primeiros artigo científicos usando a GPU para cálculos além do domínio da renderização, havia uma dificuldade na programação deste hardware para este fim. Pois as linguagens de programação existentes eram voltadas para tarefas gráficas, exigindo um amplo conhecimento sobre APIs gráficas, bem como o entendimento das funções e limitações do hardware da GPU. Além disso, o usuário era forçado a expressar os algoritmos em termos de primitivas gráficas, tais como texturas e triângulos. Como resultado, a programação de propósito geral na GPU era limitada somente aos desenvolvedores gráficos com conhecimentos avançados [5].

Uma linguagem de programação chamada de Brook surge em 2004 para tentar resolver os problemas mencionados acima sobre GPGPU. Na verdade, Brook pretendia ser um ambiente de programação com o intuito de permitir que a GPU fosse vista como um co-processador *stream*. Suporte nativo a *streams*, permitindo ao programador expressar paralelismo de dados existentes em seus programas, portabilidade e desempenho eram algumas das características desejadas na construção da linguagem Brook [5].

Em 2006 as GPUs fabricadas pela NVIDIA e pela AMD passam a adotar uma arquitetura unificada de *shaders*, arquitetura esta que apareceu pela primeira vez com o surgimento do processador Xenos da ATI. Ao invés de usar processadores específicos para diferentes etapas do *pipeline*, nesta arquitetura o processador gráfico possui uma grande quantidade de apenas um tipo de núcleo, e cada núcleo pode ser alocado dinamicamente para executar diferentes tarefas do *pipeline*. Consequentemente, o *pipeline* gráfico passa a ser uma abstração de software [30].

A arquitetura unificada *shaders* é a adotada pelas GPUs modernas. Esta estratégia torna a GPU eficiente (tanto em termos de balanceamento de carga quanto

em utilização de recursos de processamento) para executar tarefas gráficas e computacionais de propósito geral. Portanto, a adoção desta arquitetura foi fundamental para aproximar a GPU da computação de propósito geral [30].

A disparidade de desempenho entre estes dois hardwares deve-se a diferenças arquiteturais. O hardware da CPU é otimizado para alta desempenho para códigos seriais, então muitos transistores são dedicados para dar suporte a tarefas não computacionais tais como *branch prediction* e *cache*. Na outra ponta, a natureza altamente paralela das computações gráficas permite às GPUs devotarem mais transistores para computação aritmética, atingindo uma intensidade aritmética maior com o mesmo número de transistores [50].

Portanto, a GPU é mais apropriada que a CPU para resolver problemas que podem ser expressos com uma computação paralela de dados (o mesmo programa é executado em cada elemento de dado) com alta intensidade aritmética (esta intensidade é medida pela razão entre operações aritméticas e operações de memória).

A figura 4.1 ilustra duas diferenças que podem ser usadas para explicar parcialmente a superioridade do poder computacional da GPU sobre a CPU em algoritmos paralelos. Nesta figura são exibidos dois gráficos, o superior compara a evolução das capacidades da GPU e da CPU em realizar operações em ponto flutuante, de precisão simples, e o outro compara a evolução da largura de banda de acesso à memória principal.

Para controlar todo este potencial da GPU, uma plataforma de software deve ser usada. A plataforma chamada OpenCL (*Open Computing Language*) foi escolhida para implementar o mSPH em GPU. Os três principais atrativos para a adoção do OpenCL neste neste trabalho de dissertação se deve ao fato de ele ser: i) um padrão aberto; ii) multi-plataforma; e iii) capaz de rodar em dispositivos paralelos de diferentes naturezas, por exemplo GPU, CPU, FPGA etc. Uma explicação breve dos elementos conceituais desta plataforma necessários à implementação do mSPH será resumida nos próximos parágrafos.

O OpenCL é um padrão aberto capaz de proporcionar ao programador um

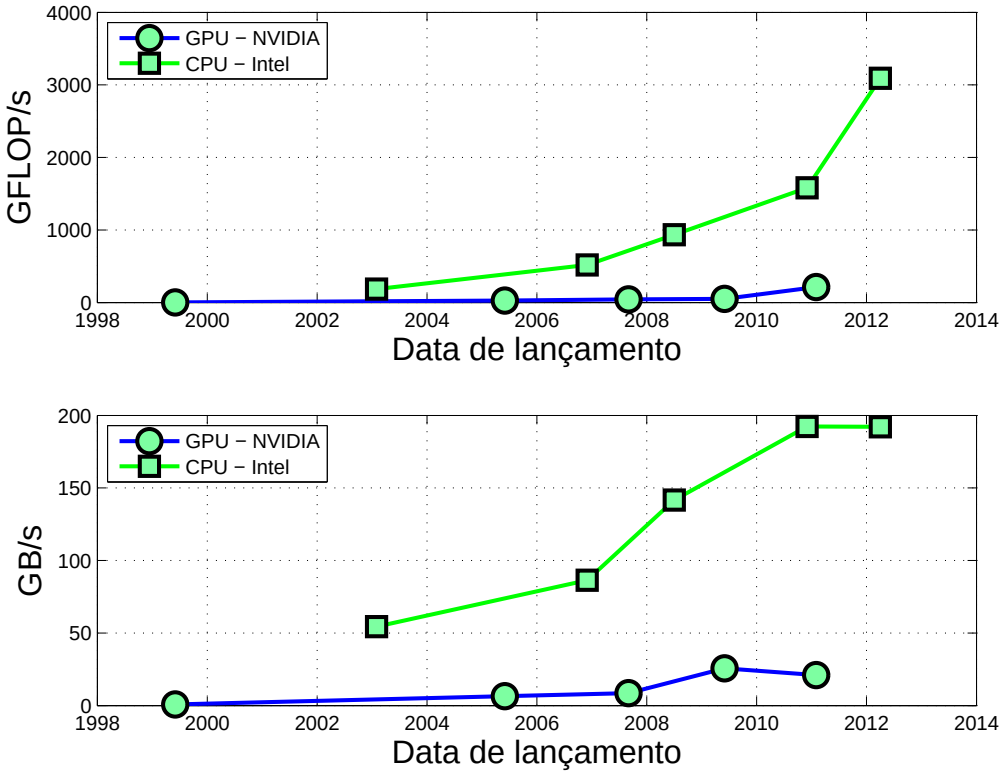

Figura 4.1: As linhas verdes e azuis correspondem à evolução do hardware, ao longo do tempo, de GPUs da NVIDIA e CPUs da Intel respectivamente. Os valores em GFLOP/s (no uso da instrução MAD = MUL+ADD) e GB/s (largura de banda de acesso à memória *off-chip*) são valores máximos teóricos aproximados e foram obtidos nos sites wwww.intel.com e www.nvidia.com. As operações em ponto flutuante são de precisão simples.

ambiente de programação de software para computação paralela. Este padrão permite programar uma coleção de dispositivos heterogêneos em uma única plataforma. Portanto, uma única plataforma pode conter, por exemplo, inúmeros CPUs e GPUs como sendo alvo de computações paralela. Mais que uma linguagem, o OpenCL é um *framework* para programação paralela que inclui API, bibliotecas e um sistema de *runtime* para dar suporte ao desenvolvimento de software. Usando o OpenCL um programador pode escrever programas de propósito geral que executam em GPUs sem a necessidade de mapear os seus algoritmos nas APIs gráficas 3D, tal tomo OpenGL.

Para entender o OpenCL se faz necessário entender a abstração de software e hardware através do uso de sua hierarquia de modelos. São eles: i) plataforma; ii) execução; iii) memória; e iv) programação. O principal modelo é o de plataforma, o qual consiste em um hospedeiro (*host*) conectado a um ou mais dispositivos computacionais (*Compute Devices*). Um dispositivo OpenCL é dividido em uma ou mais unidades computacionais (*Compute Units*), que por sua vez são divididas em um ou mais elementos de processamento (*Processing Elements*). É nestes últimos que ocorre a computação.

Uma aplicação OpenCL roda no hospedeiro de acordo com os modelos nativos da plataforma hospedeira, enviando comandos para executar computação nos elementos de processamentos dentro de um dispositivo. Os elementos de processamento dentro de uma unidade de computação executam um único *stream* de instruções como uma unidade SIMD ou como unidades SPMD (em que cada elemento de processamento mantém seu próprio contador de programa).

O modelo de execução OpenCL exige que a computação se dê em duas partes: i) *kernels* que executam em um ou mais dispositivos; e ii) um programa hospedeiro que executa no hospedeiro. É de responsabilidade do programa hospedeiro definir o contexto para os *kernels* e gerenciar a execução de cada um deles. Quando um *kernel* é submetido pelo programa hospedeiro para execução, um espaço de índices é definido. Cada ponto neste espaço corresponde à execução de uma instância de

um *kernel*, chamada de item de trabalho (*work-item*). Para a identificação de uma instância, cada item de trabalho possui um índice global único. Todos os itens de trabalho executam o mesmo código, todavia a sequência de execuções através do código e operações nos dados pode variar em cada um deles.

Um contexto é criado e manipulado pelo programa hospedeiro usando-se a API do OpenCL. Ele é composto de uma coleção de dispositivos OpenCL, *kernels*, objetos-programa e objetos-memória. Os *kernels* são funções em OpenCL que rodam nos dispositivos. Cada objeto-programa é composto por código fonte e código executável, os quais implementam um *kernel*. Objetos-memória contêm valores que podem ser usados em computações efetuadas por *kernels*. Um objeto-memória é visível tanto pelo programa hospedeiro quanto pelos dispositivos OpenCL.

A coordenação da execução dos *kernels* nos dispositivos se dá através do uso de uma estrutura de dados chamada de fila de comandos. O programa hospedeiro deve criar uma fila e associá-la a um contexto. Comandos são colocados pelo programa hospedeiro na fila de comandos para então serem escalonados para execução nos dispositivos associados a um contexto. Outros comandos podem ser entregues para a fila, tais como comandos de memória, os quais servem para transferir dados para, de ou entre objetos-memória.

A execução dos comandos em uma fila é assíncrona em relação ao hospedeiro e o dispositivo OpenCL. E, em uma fila, comandos podem executar de forma relativa entre si de maneira ordenada ou fora de ordem. Na modalidade em ordem, os comandos executam na mesma ordem que são enviados para a fila. Portanto, a execução dos comandos é serializada. Já na execução fora de ordem, os comandos são enviados em ordem para a fila, contudo não há garantia de completarem antes dos comandos seguintes.

O modelo conceitual de memória do OpenCL é composto de quatro regiões distintas. Estas regiões de memória estão acessíveis aos itens de trabalho executando um *kernel*, e se diferenciam por características de tipo de acesso (permissão de escrita/leitura), velocidade e escopo. As regiões em ordem de velocidade de acesso

crescente e escopo reduzido são: i) global; ii) constante; iii) local; e iv) privada.

A memória global é uma região onde todos os itens de trabalho, na computação de qualquer *kernel*, têm permissão de escrita e leitura em quaisquer elementos de um objeto-memória. Portando, esta região tem escopo de *kernel*. A memória constante é uma região da memória global que permanece constante durante a execução de um *kernel*. O hospedeiro é o responsável por alocar objetos na memória global e na memória constante, bem como inicializar objetos na memória constante.

A memória local é uma região compartilhada entre itens de trabalho pertencentes a um único grupo, uma vez que itens de trabalho são divididos, de forma explicita ou implicita, em grupos de trabalho (*work-group*). O objetivo então é alocar elementos que são compartilhadas por todos os itens de trabalho de um grupo de trabalho. E por último, a memória privada é uma região privada a um item de trabalho. As variáveis definidas na região privada de um item de trabalho não são visíveis por nenhum outro item. A leitura e escrita são permitidas em ambas as regiões citadas por último.

Os modelos de memória do hospedeiro e dos dispositivos OpenCL são, na maioria das vezes, independentes entre si (por exemplo, quando considerando o programa hospedeiro rodando em CPU e *kernels* rodando em GPU). Porém, estes modelos podem interagir entre si através de cópia explícita de dados de objetos-memória. A cópia explícita é feita por meio do envio de um comando de transferência de dados entre o objeto-memória e a memória do hospedeiro.

O modelo de programação paralela do OpenCL suporta paralelismo de dados. Neste modelo, define-se uma computação em termos de uma sequência de elementos pertencentes a um objeto memória. O espaço de índices associado ao modelo de execução do OpenCL define a quantidade de itens de trabalho e como se dá o mapeamento entre dados e itens de trabalho. Um modelo estritamente paralelo de dados define um mapeamento um-para-um durante a execução paralela de um *kernel*. Fica a cargo do programador estabelecer um tipo de mapeamento.

Antes de apresentar os algoritmos e suas respectivas implementações para-

lelas, cabe uma explicação importante. O modelo de computação paralela proporcionado pelo OpenCL é bastante flexível, e apenas um subconjunto deste modelo foi necessário. Por exemplo, todos os algoritmos e implementações discutidos nesta dissertação tem como foco a computação em uma única GPU usando a API do OpenCL. A plataforma de hardware usada para as implementações é composta de uma CPU e uma GPU, com suas respectivas memórias RAM (*off-chip*). O programa hospedeiro é imperativo e roda em uma única *thread* na CPU. Para enviar comando à GPU, uma única fila de comandos (ordenada) é usada. A GPU é o único dispositivo computacional OpenCL usado, sendo responsável por computar os *kernels* usando n (valor que varia de acordo com *kernel*) *threads*, e cada uma contendo seu único $id \in \{0 \leq id < n \mid id \in (\mathbb{N} \cup 0)\}$. Durante a computação na GPU, a CPU fica ociosa, aguardando o retorno para continuar a execução do programa hospedeiro. A chamada de um *kernel* se dá de forma implícita através do uso da fila de comandos, ver fig. 4.2.

O OpenCL ainda não possui uma sincronização global de *threads*, apenas *threads* pertencentes a um mesmo grupo de trabalho podem ser sincronizadas. A sincronização global ocorre de forma implícita ao término de um *kernel*, quando o controle da execução retorna ao programa hospedeiro. Portanto, é comum usar múltiplos *kernels* para realizar a computação de um único algoritmo.

4.2 Primitivas utilizadas na implementação do mSPH

4.2.1 Soma prefixal (*Scan*)

Soma prefixal, *Scan*, ou ainda *all-prefix-sum*, é uma primitiva que serve como importante bloco de construção para vários algoritmos. Uma revisão sobre o *Scan* e suas aplicações pode ser vista em [4]. Nesta dissertação, o *Scan* é primariamente usado como peça chave para a construção do algoritmo de ordenação *Radix Sorting*.

Definição: Seja um operador binário associativo $\odot$ e um conjunto ordenado

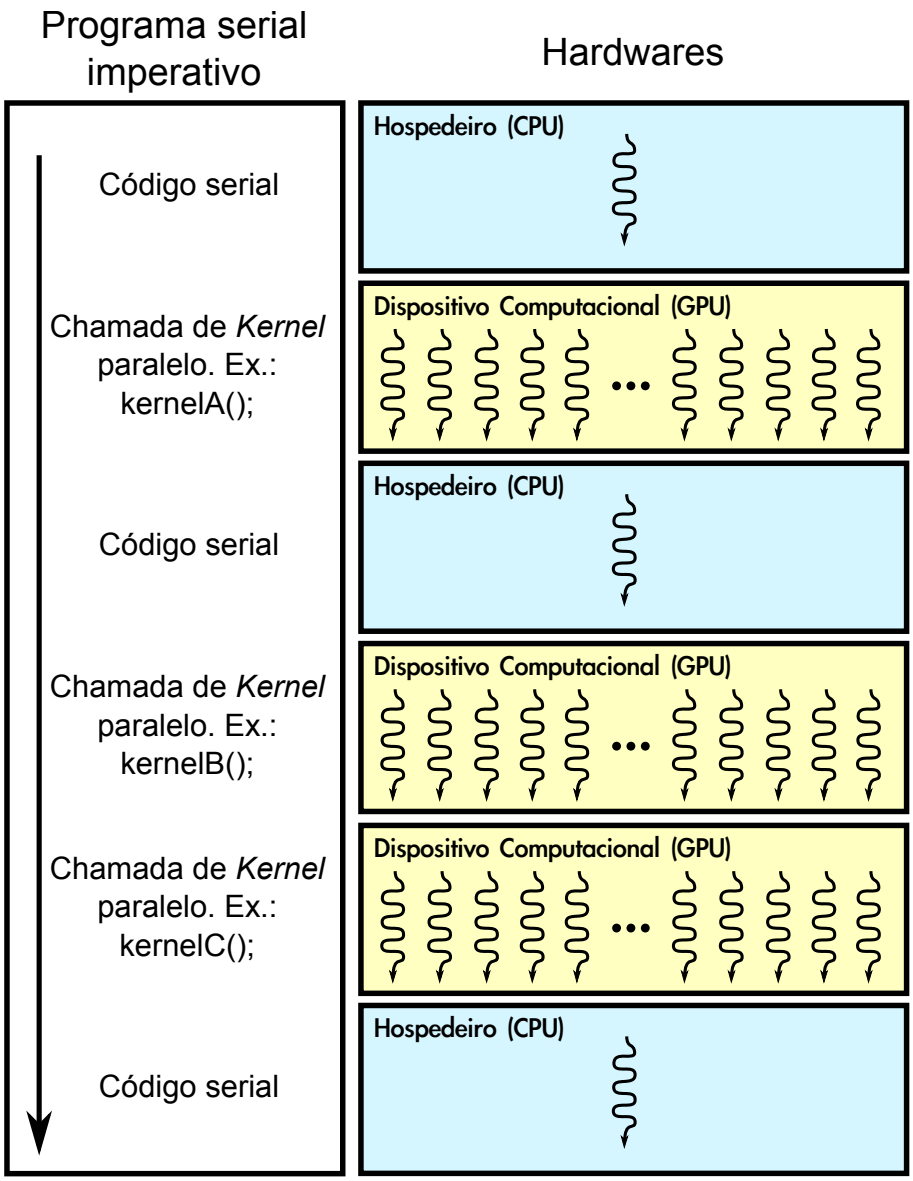

Figura 4.2: Ilustra com se dá a computação, em termos de *threads*, tanto em CPU como em GPU.

de elementos

$$[a_1, a_2, \ldots, a_n], \tag{4.1}$$

a operação *Scan* retorna o conjunto ordenado

$$[a_1, (a_1 \odot a_2), \ldots, (a_1 \odot a_2, \odot \cdots \odot a_n)], \tag{4.2}$$

Exemplo: Seja $\odot$ a operação de adição, então o *Scan* em

$$[3 \quad 1 \quad 7 \quad 0 \quad 4 \quad 1 \quad 6 \quad 3] \tag{4.3}$$

resulta em

$$[3 \quad 4 \quad 11 \quad 11 \quad 15 \quad 16 \quad 22 \quad 25] \tag{4.4}$$

Implementação: Em [34] pode-se encontrar uma implementação eficiente do *Scan* paralelo em GPU.

4.2.2 Ordenação (*Radix Sorting*)

Um dos mais antigos métodos de ordenação, o *Radix Sorting* foi implementado no início da década de 1880. Esta técnica se baseia na representação posicional de chaves, *i.e.*, cada chave é composta de uma sequência de símbolos (*i.e.*, dígitos) especificados do menos significativo até o mais significativo. Dado um alfabeto ordenado de símbolos e uma sequência de chaves como entrada, o *Radix Sorting* produz uma ordenação lexicográfica destas chaves.

Definição: O *Radix Sorting* é um algoritmo iterativo. Considerando chaves de até n dígitos, cada passo de iteração executa uma ordenação das chaves baseada na posição do i-ésimo dígito, do menos significativo ao mais significativo.

Exemplo: Seja a lista de entrada $[736, 125, 563, 274]$ de chaves, cada uma formada por 3 dígitos. Apenas um dígito em cada iteração é levado em conta, então o funcionamento do algoritmo é ilustrado por meio da figura 4.3.

Implementação: Em [35] pode-se encontrar uma implementação eficiente do *Radix Sorting* paralelo em GPU.

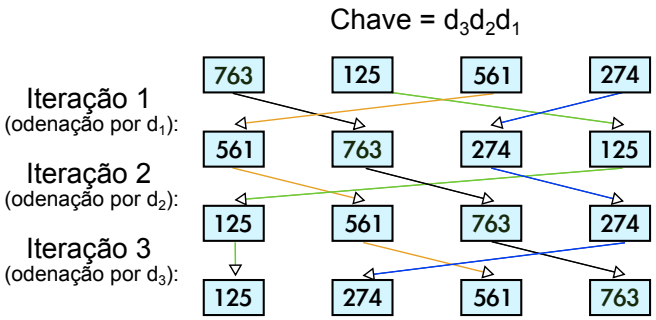

Figura 4.3: Exemplo de funcionamento do *Radix Sorting* utilizando como entrada um conjunto de chaves: $[736, 125, 563, 274]$.

4.2.3 Detecção de colisão

Muitos algoritmos de detecção de colisão (CD) são compostos de dois estágios. O primeiro, chamado de *Broad Phase* (BP), tem como entrada todos os objetos no ambiente virtual e produz uma lista de pares de objetos que estão suficientemente próximos para uma possível interseção entre suas geometrias.

É comum envolver toda a geometria, possivelmente complexa, dos objetos participantes na BP por uma geometria mais simples, tal como uma caixa ou esfera (ver fig. 4.4), chamada de *Bounding Volume* (BV). Esta técnica de invólucros tem como objetivo facilitar os testes de interseção. Se nesta primeira etapa do algoritmo é reportado que dois invólucros não se intersectam, então mais nenhum teste é necessário envolvendo o conteúdo desses invólucros.

A segunda etapa do algoritmo de CD é conhecida como *Narrow Phase* (NP). Esta etapa tem como entrada a lista produzida na BP e processa cada par de objetos para determinar o status da colisão, identificando porções das geometrias dos objetos que se intersectam.

No SPH em particular, considerando apenas o uso de invólucros esféricos na BP, a segunda etapa é dispensável, pois as partículas de fluido possuem geometria também esférica. Por este motivo, somente a etapa BP é levada em consideração nesta dissertação.

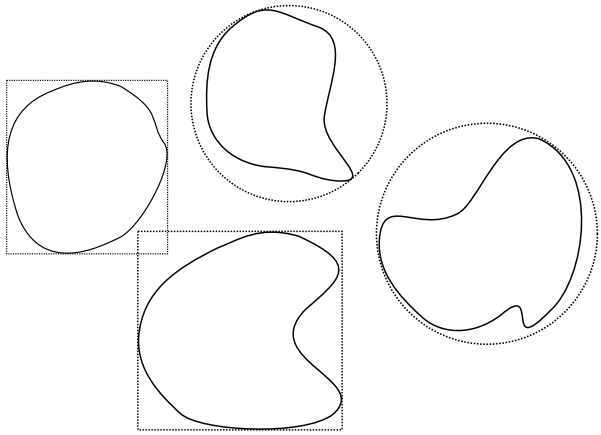

Figura 4.4: Exemplo em que os objetos estão desenhados em linha contínua e os *Bounding Volumes* em linha pontilhada.

Duas classes de algoritmos comumente adotadas para resolver a BP são: Subdivisão Espacial (SE) [14] e *Sweep and Prune* (SAP) [2, 9]. A SE consiste em procurar interseções entre objetos ocupando a mesma região espacial. O SAP é uma técnica baseada em ordenação da projeção da extensão dos invólucros em cada eixo cartesiano.

Em [25] foi proposto um algoritmo híbrido, contendo tanto o SAP quanto o SE, concebido para rodar em GPU. Este artigo reportou que o algoritmo implementado pelos autores de [25] era até 71 vezes mais rápido que um algoritmo considerado como o estado da arte, este último rodando em CPU, e até 212 vezes mais rápido quando comparado a outros métodos que rodavam em GPU.

O algoritmo de CD, chamado a partir deste ponto de CD-SPH, usado nesta dissertação para implementar o mSPH é derivado de [25]. A observação das características das simulações SPH permitiu refinar o CD proposto em [25], criando um CD especializado que é mais simples de implementar.

As etapas que contemplam o CD-SPH podem ser vistas no fluxograma dado pela figura 4.5. Excluindo-se o estado de decisão e o imediatamente anterior a este, todos são executados em GPU. Cada estado pode fazer uso de uma ou mais

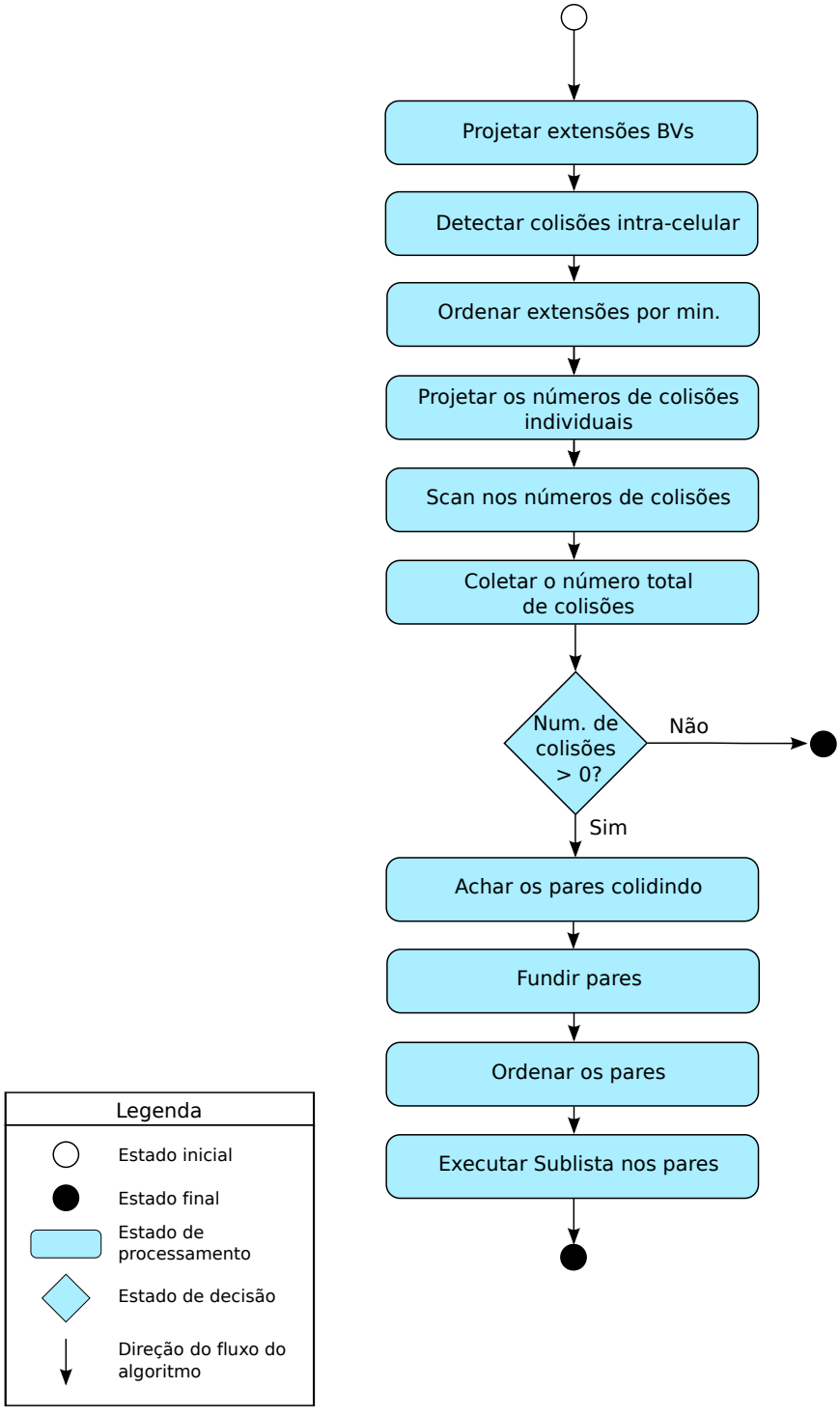

Figura 4.5: Fluxograma do algoritmo de colisão CD-SPH.

estruturas de dados, as quais devem ser alocadas na memória principal da GPU uma única vez e antes da primeira execução do CD-SPH.

Três constantes são usadas para calcular o tamanho das estruturas acima mencionadas e, da mesma forma que estas, as constantes também devem ser armazenadas na memória principal da GPU. Estas constantes são exibidas na tabela 4.1.

k	Número de partículas na simulação.
α	Número máximo de colisões que cada thread pode detectar envolvendo uma única partícula.
n	Número máximo de colisões que podem ocorrer em um único passo de tempo, considerando todas as partículas. Este valor é calculado como $n = \alpha k$.

Tabela 4.1: Tabela contendo constantes usadas no CD-SPH.

Um total de nove vetores são usados como estruturas: $A1$, $A2$, C, D, L, S, O, $\in \mathbb{R}^n$ e $P1$, $P2 \in \mathbb{R}^{2n}$. Cada um destes vetores armazena apenas um número em ponto flutuante na i-ésima posição, com a exceção do $A1$ que armazena dois.

A construção do algoritmo pressupõe que a simulação ocorra em um domínio $\Omega \subset [0,1] \times [0,1]$ e que todas as k (a_0, a_1, ... , a_{k-1}) partículas, contidas em Ω, são círculos de mesmo raio r. Como consequência deste fato, o CD-SPH espera receber como entrada uma variável escalar k contendo o número total de partículas na simulação, uma outra variável r, também escalar, com o raio das partículas e, por último, uma estrutura vetorial $X \in \mathbb{R}^k$ contendo a posição das partículas.

As variáveis de entrada e saída do CD-SPH estão descritas na tabela 4.2. Para exemplificar a saída do CD-SPH, considere o conjunto de partículas dado pela figura 4.6. A saída associada à entrada é dada pela figura 4.7.

A descrição do CD-SPH detalhada é dada a seguir, explicando-se a tarefa de cada estado:

1. **Projetar extensões BVs**: A extensão do BV de uma partícula a quando projetada no eixo x permite obter um intervalo de extremos unidimensional

Entrada	
k	Número total de partículas na simulação.
X	Vetor contendo as posições das partículas.
r	Raio das partículas.
Saída	
η_L	Número de sublistas em L.
L	Armazena o início das sublistas, *i.e.*, a i-ésima posição deste vetor guarda o índice do primeiro elemento da i-ésima sublista armazenada em $P1$ (ver seção 4.2.4).
S	Armazena o tamanho das sublistas, *i.e.*, a i-ésima posição deste vetor guarda o tamanho da i-ésima sublista armazenada em $P1$ (ver seção 4.2.4).
$P1$	Armazena o primeiro elemento dos pares de colisão. Estes elementos são agrupados em sublistas (ver seção 4.2.4).
$P2$	Armazena o segundo elemento dos pares de colisão.

Tabela 4.2: Descreve os conjunto de variáveis de entrada e saída.

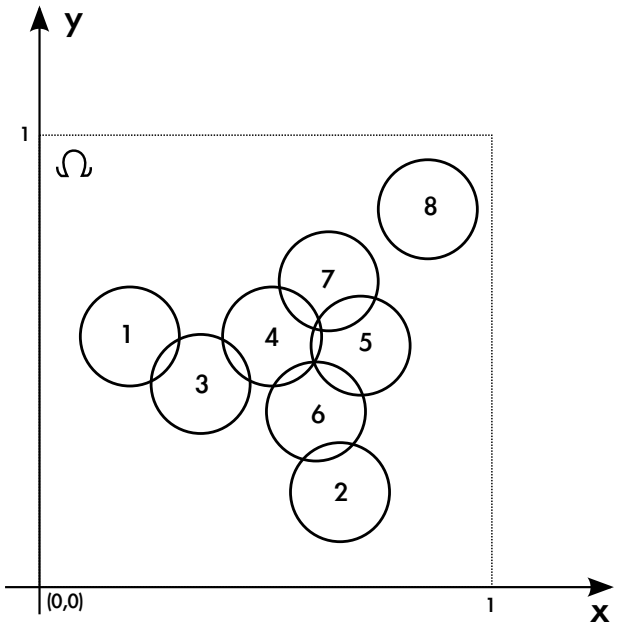

Figura 4.6: Conjunto de partículas em um domínio usado para exemplificar a saída do CD-SPH.

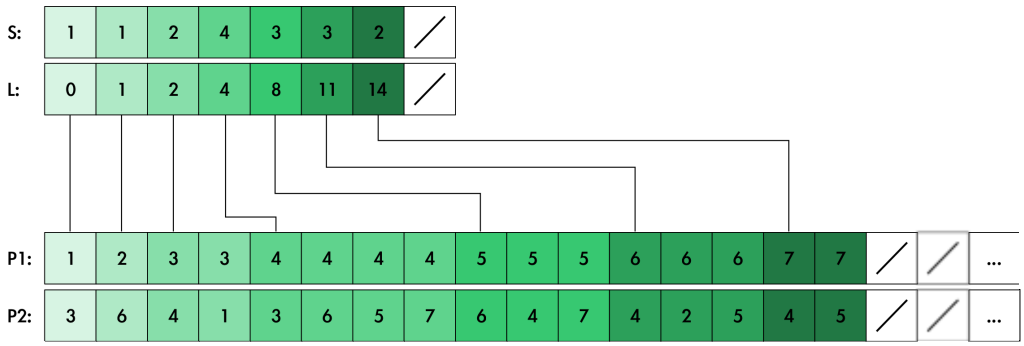

Figura 4.7: Exemplo de saída correspondente à entrada dada pela figura 4.6

.

$\tau_a = [m_a, M_a]$, em que $m_a = x_a - r$, $M_a = x_a + r$, sendo r o raio do BV (ver figura 4.8). Esta etapa consiste em obter os intervalos τ_{a_i}, $0 \leq i < k$, para preencher os vetores $A1$ e $A2$, fazendo $A1[i] = \{M_{a_i}, a_i\}$ e $A2[i] = m_{a_i}$. Para alcançar tal objetivo, basta fazer uso de um *kernel* onde cada *thread* faz a projeção de uma ou mais partículas.

2. **Detectar colisões intra-celular**: Com o intuito de fazer a parcela da SE, esta etapa consiste em dividir o domínio em uma grade regular de $4 \times 4 = 16$ células, particionando o eixo x em 4 divisões e o eixo y também em 4 divisões. A cada partícula se associa uma palavra de, pelo menos, 24 bits. O bit $i = 1$ se a partícula intersecta a célula i, caso contrário, o bit $i = 0$. As palavras são armazenadas no vetor D. A execução desta tarefa, em termos de número de *threads*, ocorre da mesma forma que a tarefa anterior.

3. **Ordenar extensões por mínimo**: Ordenar os pares $\langle chave_a, valor_a \rangle = \langle m_a, \{M_a, a\} \rangle$, portanto ordenar os elementos dos vetores $A2$ e $A1$, usando a primitiva *Radix Sorting*.

4. **Projetar os números de colisões individuais**: Esta etapa consiste em contar, dado um intervalo $\tau_a = [m_a, M_a]$ pertencente a uma partícula a, o número η_a de intervalos $\tau_b = [m_b, M_b]$, tal que $m_a < m_b < M_a$, ou seja, os

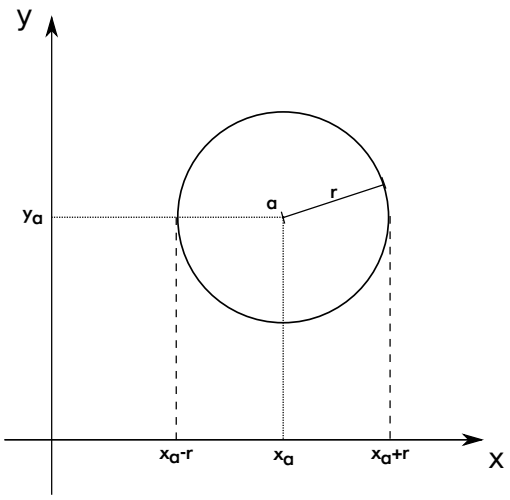

Figura 4.8: Projeção da extensão de uma partícula a qualquer no eixo x.

intervalos τ_b que "sobrepõem" e estão à "direita" de τ_a. O resultado desta etapa fica em um vetor C, em que $C[a_i] = \eta_{a_i} \leq \alpha$ e $0 \leq i < k$. O *kernel* projetaNumColisoes() (pseudo-código do kernel no apêndice) realiza esta etapa através da execução de k *threads*.

5. **Scan nos números de colisões**: Usando-se em C a soma prefixal (*scan*), exclusiva e em paralelo, mencionada na seção 4.2.1, obtém-se um vetor de deslocamentos (*offsets*) O.

6. **Coletar o número total de colisões**: Seja λ o número total de colisões, então este número é obtido fazendo $\lambda = C[k-1] + O[k-1]$. Esta etapa ocorre no programa hospedeiro.

7. **Achar os pares colidindo**: Esta etapa é semelhante à etapa 4, porém em vez de contar os pares, coleta-os. Para isto, considere a mesma matriz $A_{2,n}$ mencionada na seção (4.2.4), aqui representada pelos vetores $P1$ e $P2$, para também guardar os pares de colisão. O objetivo desta tarefa é preencher $P1$ e $P2$ em paralelo, usando k *threads*, uma para cada partícula. Seja a partícula $a = A1[id]$ e $\eta_a = C[a]$, então este objetivo é alcançado fazendo $P1[i] = a$ e

$P2[i] = A1[id+l]$, em que $i = O[a]+l$ e $0 \leq l < \eta_a$, onde id representa o índice de cada uma das *threads*. O *kernel* achaParesColidindo() (pseudo-código do kernel no apêndice) realiza esta etapa através da execução de k *threads*.

8. **Fundir pares**: Esta etapa duplica os pares em $P1$ e $P2$, *i.e.*, para cada par $\phi_{ab} = (a, b)$, cria-se o par $\phi_{ba} = (b, a)$. Para tal tarefa, basta fazer $P1[\lambda+id] = P2[id]$ e $P2[\lambda+id] = P1[i]$, $0 \leq id < \lambda$, em que id representa o índice de cada uma das λ *threads* que devem ser executadas em um *kernel*.

9. **Ordenar os pares**: Todos os pares de colisão estão neste momento em $P1$ e $P2$ de forma desordenada, portanto se faz necessário ordenar os elementos de $P1$ e $P2$ usando *Radix Sorting*, em que $\langle chave_a, valor_a \rangle = \langle P1[a], P2[a] \rangle$ e $0 \leq a < 2\lambda$.

10. **Executar Sublista nos pares**: O algoritmo Sublista é detalhado na seção (4.2.4), contendo a motivação para usá-la nesta etapa. O Sublista deve ser feito em $P1$, produzindo como saída os vetores L e S, assim como a variável η_L (ver fig. 4.7).

4.2.4 Sublistas em um vetor ordenado com repetição (Sublistas)

Para descrever esta primitiva, chamada nesta dissertação de Sublistas, pode-se usar a seguinte motivação: no contexto de implementação computacional do SPH, considere a matriz $A_{2,n} = [a_{ij}]$ na qual a coluna $C_j = [a_{1j}\, a_{2j}]^T$ expressa que a partícula a_{1j} interage com uma outra partícula a_{2j}, *i.e.*, C_j é um par de interação entre partículas e, portanto, $A_{2,n}$ contém n pares de interação.

Dado que $a_{11} \leq a_{12} \leq \ldots \leq a_{1n}$, o problema consiste em achar o índice da primeira ocorrência (da esquerda para a direita) de cada partícula na primeira linha da matriz $A_{2,n}$ e o número de ocorrências destas (também somente na primeira linha). A figura 4.9 ilustra a motivação através de um exemplo.

Definição: Seja um conjunto S ordenado e com repetição de n números distintos, e definindo uma sublista como um conjunto $s_i \in S$ de k_i termos iguais

Figura 4.9: Ilustração da motivação proposta para usar o algoritmo Sublistas. Exemplo de uma entrada, contendo pares de interação entre partículas, e uma possível saída.

$a_1^i = a_2^i = \ldots = a_{k_i}^i$, esta primitiva consiste em achar as sublistas $s_1, s_2, \ldots, s_i | 1 \leq i \leq n$ e $\sum k_i = n$. O vetor S, portanto, é da forma

$$S = [\ \underbrace{a_1^1\ a_2^1\ \ldots\ a_{k_1}^1}_{s_1}\ \underbrace{a_1^2\ a_2^2\ \ldots\ a_{k_2}^2}_{s_2}\ \ldots\ \underbrace{a_1^n\ a_2^n\ \ldots\ a_{k_n}^1}_{s_n}\ ]. \tag{4.5}$$

O algoritmo tem como saída um vetor I contendo a posição de cada sublista s_i, *i.e.*, o índice do primeiro elemento a_1^i em S, um outro vetor Q contendo a quantidade de termos em cada sublista s_i e um escalar contendo a quantidade de sublistas encontradas.

Exemplo: Seja a entrada dado por

$$S = [1 \quad 1 \quad 2 \quad 3 \quad 4 \quad 4 \quad 4 \quad 5 \quad 5], \tag{4.6}$$

as saídas são:

$$I = [0 \quad 2 \quad 3 \quad 4 \quad 7 \quad -1 \quad -1 \quad -1 \quad -1] \tag{4.7}$$

e

$$Q = [2 \quad 1 \quad 1 \quad 3 \quad 2 \quad -1 \quad -1 \quad -1 \quad -1]\ . \tag{4.8}$$

O valor -1 indica uma posição inválida no vetor. Portanto, há neste exemplo 5 sublistas, $s_1 = \{0; 2\}$, $s_2 = \{2; 1\}$, $s_3 = \{3; 1\}$, $s_4 = \{4; 3\}$ e $s_5 = \{7; 2\}$, em que o par $\{a; b\}$ representa a posição (a) e o número de elementos (b) de uma lista qualquer, $s \in S$.

Implementação: A implementação paralela deste algoritmo faz uso de dois vetores, M e O, ambos de tamanho n, além dos vetores S, I e Q já mencionados. O vetor M, através do uso de *flags*, indica o início de cada sublista. O vetor Q serve para dar o deslocamento (*offset*) da posição que uma *thread* usa para escrever em I e Q.

O algoritmo 1 descreve a computação da primitiva Sublistas. Considere que todos os vetores necessários são previamente alocados em momento anterior a execução das funções, todos de tamanho k, e todas as funções devem executar $a \geq k$ *threads*.

```
1 Algorithm Sublistas()
    Input: k /* Número de elementos em S */
    Input: S /* Vector contendo as sublistas */
    Output: I /* Vector contendo a pos. inicial das sublistas */
    Output: Q /* Vector contendo a qtd. elem. nas sublistas */
    Output: numSublistas /* Guarda o qtd. de sublistas */
    Data: M /* Vetor de flags */
    Data: O /* Vetor com os offsets */
2   M = calculaFlags(S, k);
3   O = scanInclusivo(M, k);
4   I = calculaInicioSublistas(M, O, k);
5   Q = calculaQtdElemSublistas(S, M, k);
6   numSublistas = O[k − 1];
7 end
```

Algoritmo 1: Sublistas - A implementação do algoritmo Sublista é composta por quatro *kernels*, todos estão descritos no apêndice, exceto o scanInclusivo().

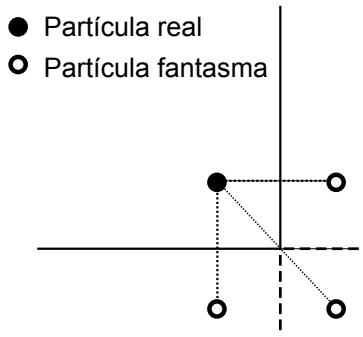

Figura 4.10: Uma partícula real próxima o suficiente da quina da fronteira gera até três partículas fantasmas.

4.3 Uma implementação computacional do mSPH

Como foi dito na introdução deste capítulo, o modelo OpenCL de programação na GPU prevê o uso da CPU como gerenciador da computação na GPU. A implementação do mSPH segue esse modelo, enviando toda computação paralela para a GPU, porém computações seriais permanecem na CPU. As estruturas de dados usadas pela GPU são estáticas e portanto previamente, *i.e.*, no início do algoritmo, alocadas na memória global da GPU. As tabelas 7.1 e 7.2 (contidas no apêndice) exibem estas variáveis.

Devido ao fato da geometria da fronteira impermeável ser retangular, cada partícula pode gerar até três partículas fantasmas (ver fig. 4.10). Portanto, as estruturas devem ser alocadas com espaço suficiente para armazenar informações de $4 \times n_r$ partículas.

Para facilitar o entendimento da implementação do mSPH, duas descrições do algoritmo completo são apresentadas. Cada descrição desta possui um foco diferenciado, porém se complementam. As imagens 4.12 e 4.13 servem para acompanhar cada descrição desta, pois explicitam a forma de armazenamento das estruturas principais necessárias a implementação do mSPH. Para o armazenamento destas estruturas optou-se por usar o *layout* SoA (*Structure-of-Arrays*) no lugar de AoS (*Array-of-Structures*) por refletir, geralmente, em melhor desempenho na arquite-

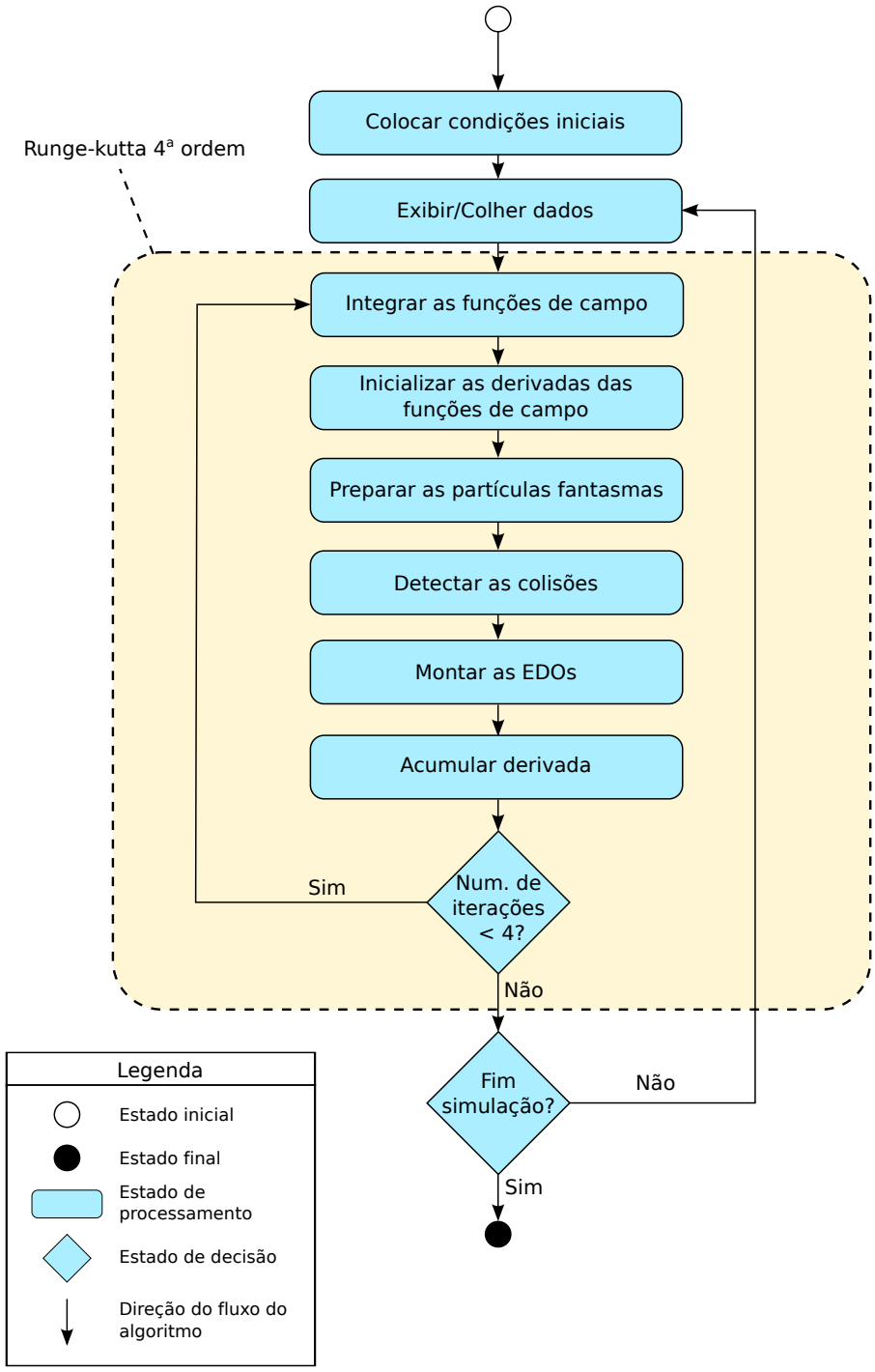

Figura 4.11: Fluxograma da imprementação do mSPH.

tura da GPU [55]. Cada array é composto de $4n_r$ vetores de tamanho 4. Esta escolha utiliza mais memória, entretanto é mais adequada, em termos de desempenho, para arquiteturas VLIW [55].

A primeira descrição chama atenção para o fato de que é necessário o dobro de armazenamento para as estruturas ($buffers$) que armazenam as funções de campo do fluido (variável q) e as variações destas (variável Q). Dois motivos contribuem para tal aumento de armazenamento. Um está relacionado à necessidade de armazenar valores nas estruturas para diferentes intervalos de tempo. O outro motivo é o problema de concorrência de escrita e leitura, pois cada estrutura desta é acessada por várias *threads* ao mesmo tempo quando executando um *kernel*. Cada variável é instanciada através de dois $buffers$, sendo um primário e outro secundário. O algoritmo 2 em pseudo-código faz uso destes $buffers$ e reflete esta descrição.

A segunda descrição é composta do fluxograma dado pela figura 4.11, mais uma explicação textual. Esta última descreve cada etapa contida no fluxograma e, quando necessário, faz uso de uma abordagem mais baixo nível, descrevendo funções e *kernels* em pseudo-código. As etapas são dadas a seguir:

1. **Colocar condições iniciais**: Primeira etapa do algoritmo, executada apenas uma única vez. E, quando comparada as outras etapas (sem levar em conta a segunda etapa), possui pouco custo computacional. Estes dois motivos permitem simplificar a sua implementação, uma vez que a codificação se dá integralmente em CPU.

2. **Exibir/Colher dados**: Dependendo do objetivo da aplicação a rodar a implementação do mSPH, pode-se aproveitar a presença dos dados (através das variáveis) da simulação na memória da GPU. Para renderizar o fluido, por exemplo, basta utilizar a API do OpenCL para interoperar com APIs gráficas, tal como OpenGL. Outras possíveis finalidades para esta estapa são, por exemplo, armazenar as variáveis de estado a cada iteração, permitindo plotar gráficos, fazer análises de erro e etc.

```
1  Algorithm mSPH()
2      fimSimulacao = false;
       // Inicializa índices para os buffers.
3      b_{k-1} = 0;  // b_{k-1} contém o índice do buffer primário.
4      b_k = 1;    // b_k contém o índice do buffer secundário.
5      b_idx = 0;  // b_idx contém o índice do buffer primário.
6      colocarCondicoesIniciais( q[b_{k-1}], Q[b_idx]);
       // Inicia a simulação.
7      do
8          exibirColherDados( q[b_{k-1}], Q[b_idx]);
9          b_idx = 1;
10         for (i = 0; i < 4; i++) do
               // Calculo de índices.
11             b_i = (6 >> i) & 1;  // Retorna 0,1,1 e 0 para i = 0..3.
12             b_d = (7 >> i) & 1;  // Retorna 1,1,1 e 0 para i = 0..3.
               // O kernel abaixo efetua a seguinte computação:
               // q[b_k][id] = q[b_{k-1}][id] + Q[b_i][id] * ΔT[i]
13             integrarFuncoes(q[b_k], q[b_{k-1}], Q[b_i], ΔT[i]);
               // O kernel abaixo efetua a seguinte computação:
               // Q[b_d][id] = {q[b_k][id].U, g, 0}
14             inicializarDerivadas(Q[b_d], q[b_k]);
15             prepararParticulasFantasmas(q[b_k]);
16             detectarColisoes(q[b_k]);
17             montarEDOs(Q[b_d], q[b_k]);
               // Acumula derivada
18             if  i < 3 then
                   // O kernel abaixo efetua a seguinte computação:
                   // Q[0][id]  += wQ[i] * Q[1][id]
19                 acumularDerivada(Q[0], wQ[i], Q[1]);
20             end
21         end
           // Troca de buffers
22         temp = b_k;
23         b_k = b_{k-1};
24         b_{k-1} = temp;
           // Restaura índice do buffer.
25         b_idx = 0;
26     while !fimSimulacao;
27 end
```

Algoritmo 2: mSPH() - Algoritmo criado para implementar o mSPH. O fato das variáveis serem passadas como argumentos para um *kernel*, ou função, serve apenas para evidenciar a importância destas variáveis na computação paralela.

3. **Integrar as funções de campo**: Seja a i-ésima iteração do Runge-Kutta de quarta ordem, em que $0 \leq i < 4$, executar a seguinte computação em um *kernel*:

$$q^n[id] = q^{n-1}[id] + Q[id] * \Delta T[i] \tag{4.9}$$

4. **Inicializar as derivadas das funções de campo**: Como $Q \equiv \{V^n, A^n, R^n\}$, para inicializar a variação temporal de cada função de campo, basta utilizar um *kernel* com a seguinte computação:

$$Q^n[id] = \{q^n[id].U, g, 0\} \tag{4.10}$$

5. **Preparar as partículas fantasmas**: A cada iteração do algoritmo, devem-se criar partículas fantasmas para impor a condição de fronteira impermeável. Cada partícula real pode gerar até três fantasmas, pois a geometria do domínio é retangular. Os passos usados consistem em contar o número total de partículas fantasmas, achar o índice pertencente a cada partícula fantasma, usado para guardar suas informações no vetor q (para isto utiliza-se uma soma prefixal no vetor N_f), preencher o vetor q com as informações das partículas fantasmas e atualizar o número total de partículas n_t. Esta etapa é descrita pela função preparaFantasmas(), a qual utiliza cinco *kernels*.

6. **Detectar as colisões**: Esta etapa existe para achar todos os pares de interação entre partículas. Estes pares são achados através do uso do algoritmo descrito na seção 4.2.3.

7. **Montar as EDOs**: Usando a informação de colisão gerada na etapa anterior, a montagem das EDOs ocorre computando-se o KI para cada partícula real. Como cada partícula possui uma lista com as partículas que ela interage, basta cada *thread* percorrer uma lista desta serialmente. Portanto a computação é feita por t *threads*, sendo t o número de sublistas retornado pela etapa de detecção de colisão.

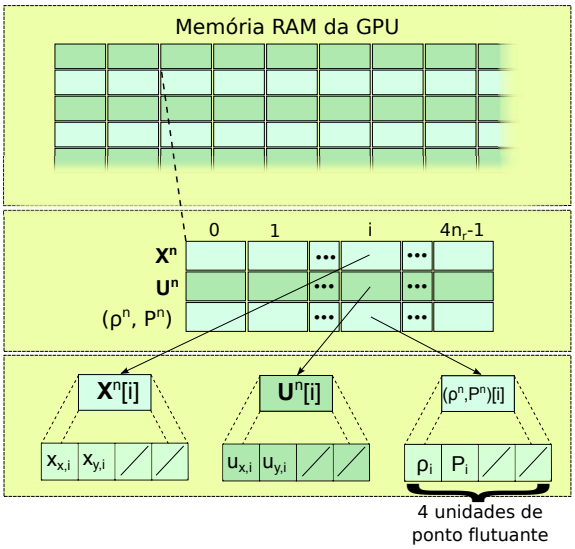

Figura 4.12: A estrutura, de tamanho fixo, q armazena a posição, a velocidade, a densidade e a Pressão de cada partícula de fluido.

8. **Acumular derivada**: O método de integração Runge-Kutta de 4ª ordem consiste em aplicar uma média de derivadas computadas durante um passo de tempo (ver sec. 3.2.5). Este passo acumula a contribuição destas derivadas.

A implementação descrita acima foi testada em uma única placa de vídeo Radeon HD6790 (esta placa possui 800 processadores *Stream* à 840MHz e 1GB de memória RAM à 4.2Gbps, permitindo um desempenho computacional máximo teórico de 1.34 TFLOPs) usando o sistema operacional Ubuntu 12.04 LTS 64bits. O programa hospedeiro foi escrito em C++. Para testar a desempenho desta implementação utilizou-se resultados provenientes de [6]. No artigo citado, o autor afirma que a sua implementação leva 15 minutos (a duração real é de 2 segundos) em um *cluster* de 4 processadores no caso de teste *dam-break* com 5000 partículas de fluido. A implementação rodando na placa Radeon roda em 3.7 minutos para o mesmo caso de teste.

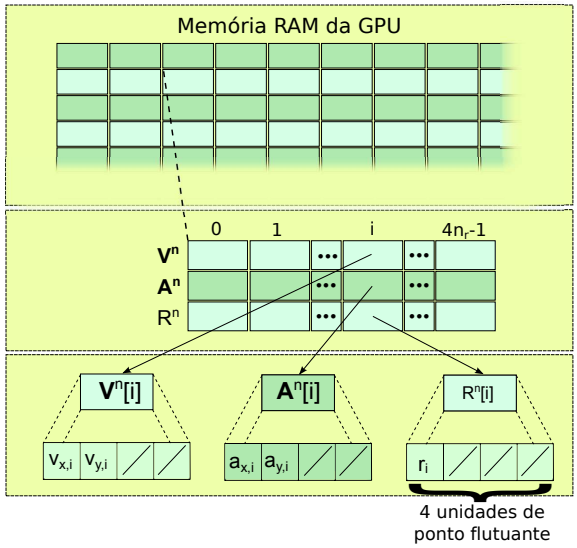

Figura 4.13: O armazenamento da estrutura Q em memória se dá de maneira análoga à q (ver fig. 4.12), todavia Q armazena as variações das funções de campo do fluido.

5 SIMULAÇÃO NUMÉRICA COM O MÉTODO MSPH

5.1 Introdução

O caso de teste usado para validar a implementação do mSPH é o mesmo descrito em [58] e possui configuração inicial esquematizada na fig. 5.1. A finalidade do caso de teste é simular um rompimento de barragem (*dam-break*). Trata-se de uma simulação bidimensional de fluido altamente não linear. Apesar da aparente simplicidade, o problema não possui solução analítica conhecida, portanto deve-se usar métodos numéricos para tentar esclarecer dúvidas sobre a evolução do fluido. Algumas aplicações relacionadas ao caso de teste citado são descritas em [10]. Este caso de teste é considerado padrão em simulações SPH que tentam validar as formulações para fluxos com superfície livre.

5.2 Caso de teste: Quebra de barragem (*dam-break*)

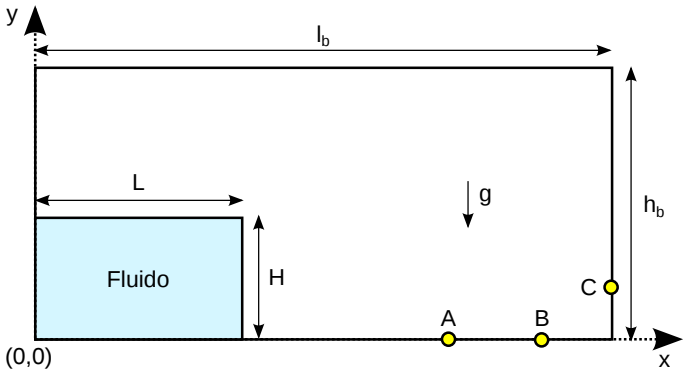

Figura 5.1: Desenho esquemático para descrever o caso de teste quebra de barragem [58].

Na fig. 5.1 pode-se ver uma massa de água retangular de dimensões $H \times L$ em repouso. O fluido está confinado em uma caixa, também retangular, de dimensões

$h_b \times l_b$. As medidas são: $H = 1m$, $L = 2H, l_b = 5.366H$ e $h_b = 0.65l_b$.Os pontos $A = (3.721H, 0)$ e $B = (4.542H, 0)$ são usados para medir a altura máxima do fluido, já o ponto $C = (5.366H, 0.192H)$ representa um sensor de pressão. Os três pontos permitem fornecer dados para comparar resultados experimentais (ver figuras 5.28 - 5.30).

O conjunto de figuras 5.2-5.26 exibe diferentes instantes de tempo das duas simulações de rompimento de barragem. A sequência de imagens permite observar a evolução temporal das velocidades vertical e horizontal, acelerações horizontal e vertical e a pressão do fluido para o mSPH sem suavização (sequência da esquerda) e com suavização (sequência da direita) em $n = 5$ passos (ver seção 3.2.6).

Os parâmetros das duas simulações são idênticos, exceto pelo fato de que uma utiliza a suavização das funções de campo e a outra não. Para as duas simulações, a velocidade artificial do som é $c = 26.46m/s$, o comprimento de suavização é $h = 0.025$, o espaçamento das partículas é $dx = 0.75h$ e a condição de Courant é $\mu_c = 0.8$. Os instantes de tempo estão em ordem crescente de cima para baixo. Destaca-se que as duas simulações possuem a mesma condição inicial, empregam os mesmos parâmetros de discretização temporal e espacial, e utilizam o mesmo esquema de cor.

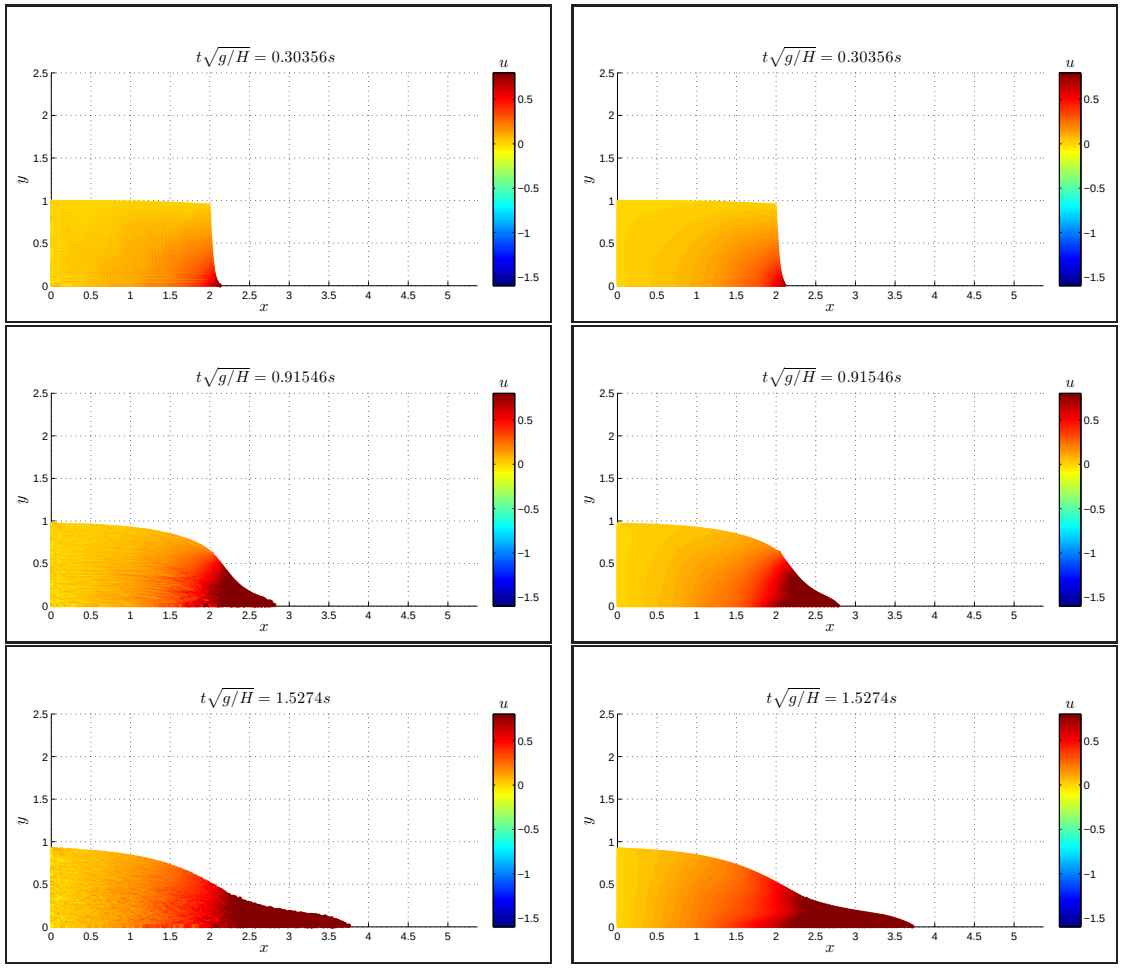

Figura 5.2: velocidade horizontal a) sem suavização e b) com suavização aplicada em $n = 5$ passos, em $t = 0.30356s$, $t = 0.91546s$ e $t = 1.5274s$.

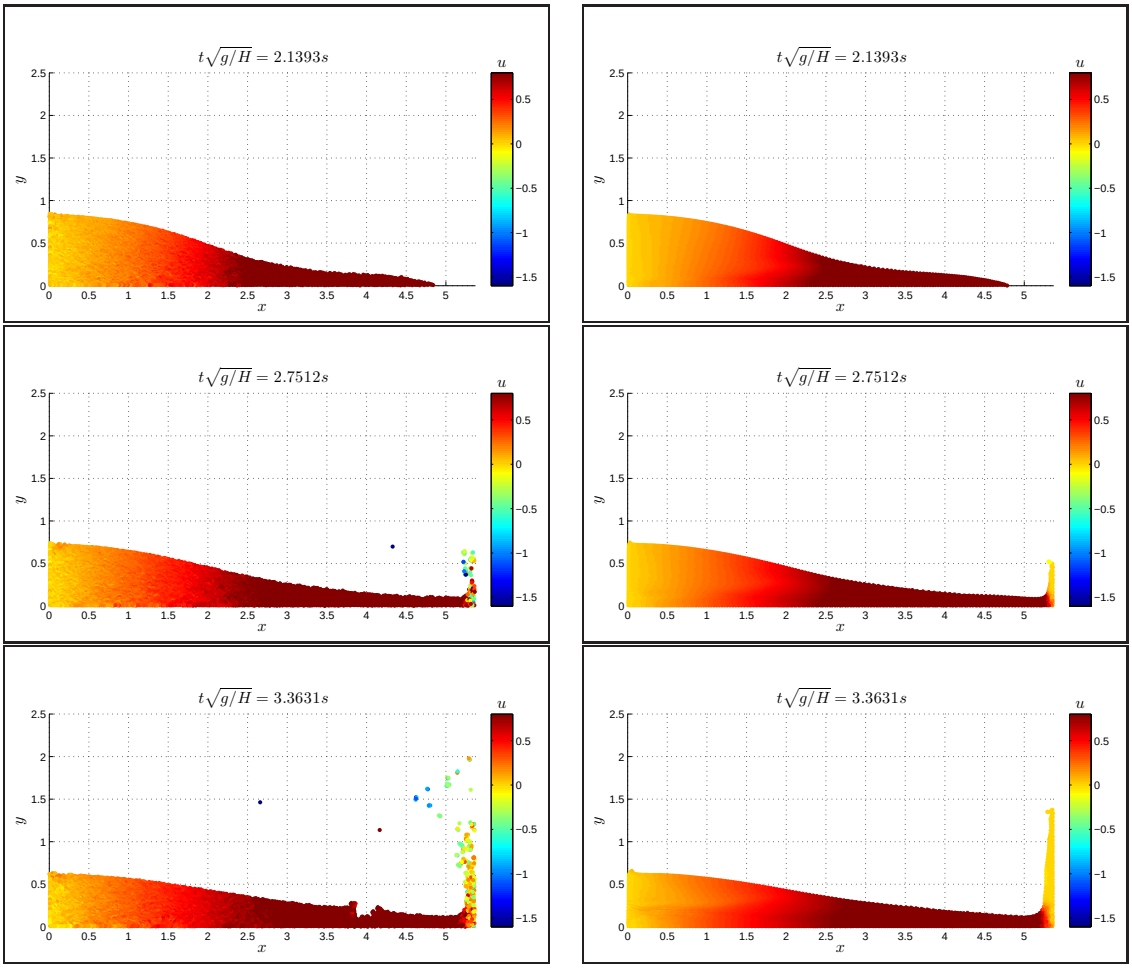

Figura 5.3: Velocidade horizontal (a) sem suavização e (b) com suavização aplicada em n = 5 passos, em $t = 2.1393s$, $t = 2.7512s$ e $t = 3.3631s$.

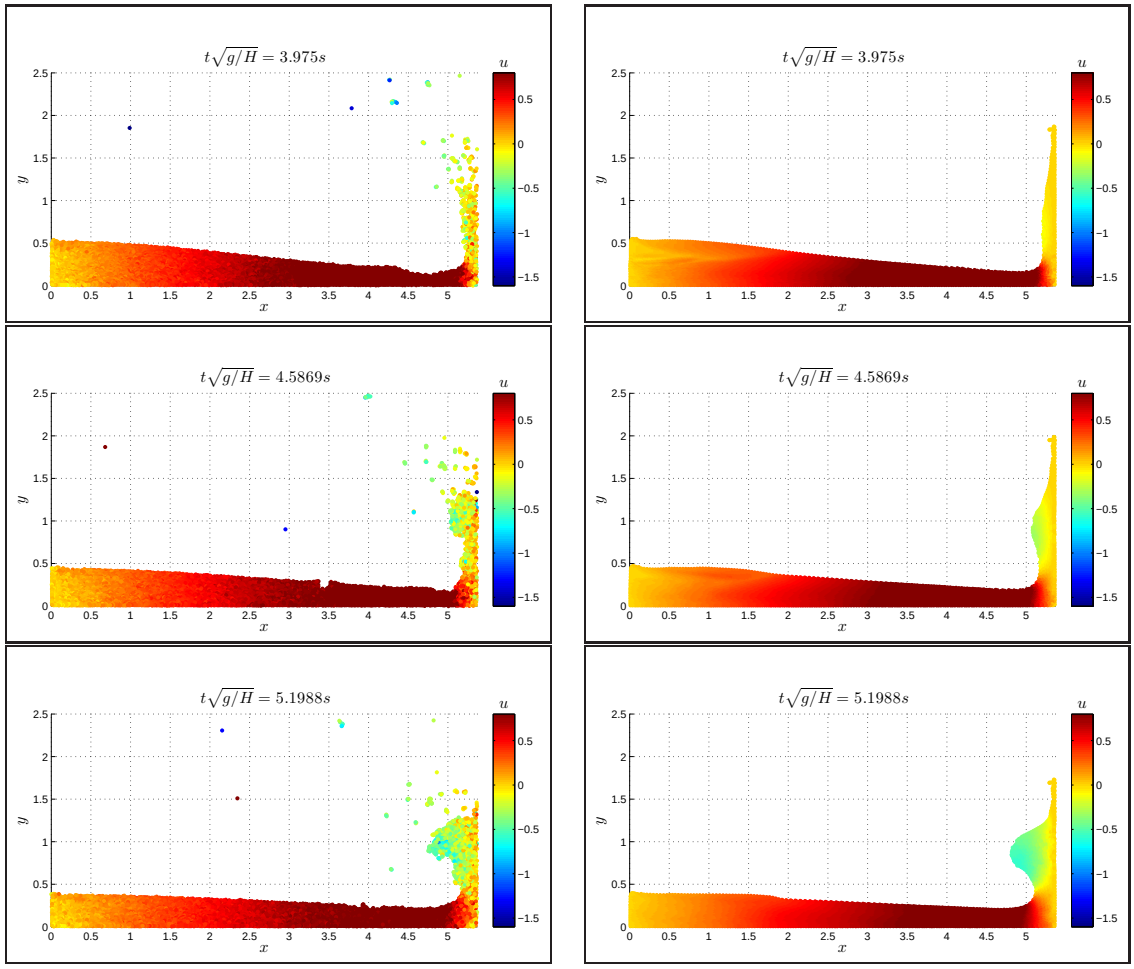

Figura 5.4: Velocidade horizontal (a) sem suavização e (b) com suavização aplicada em n = 5 passos, em $t = 3.975s$, $t = 4.5869s$ e $t = 5.1988s$.

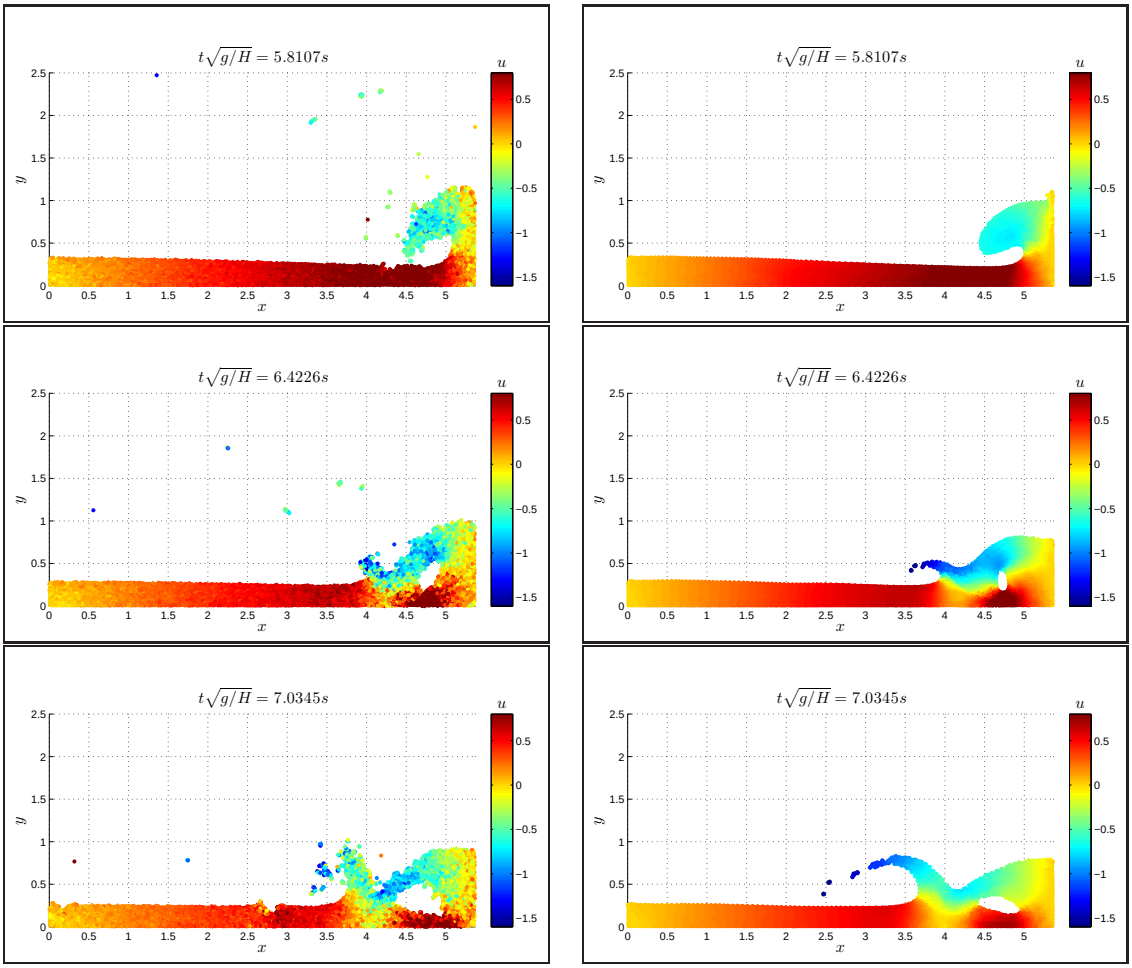

Figura 5.5: Velocidade horizontal (a) sem suavização e (b) com suavização aplicada em n = 5 passos, em $t = 5.8107s$, $t = 6.4226s$ e $t = 7.0345s$.

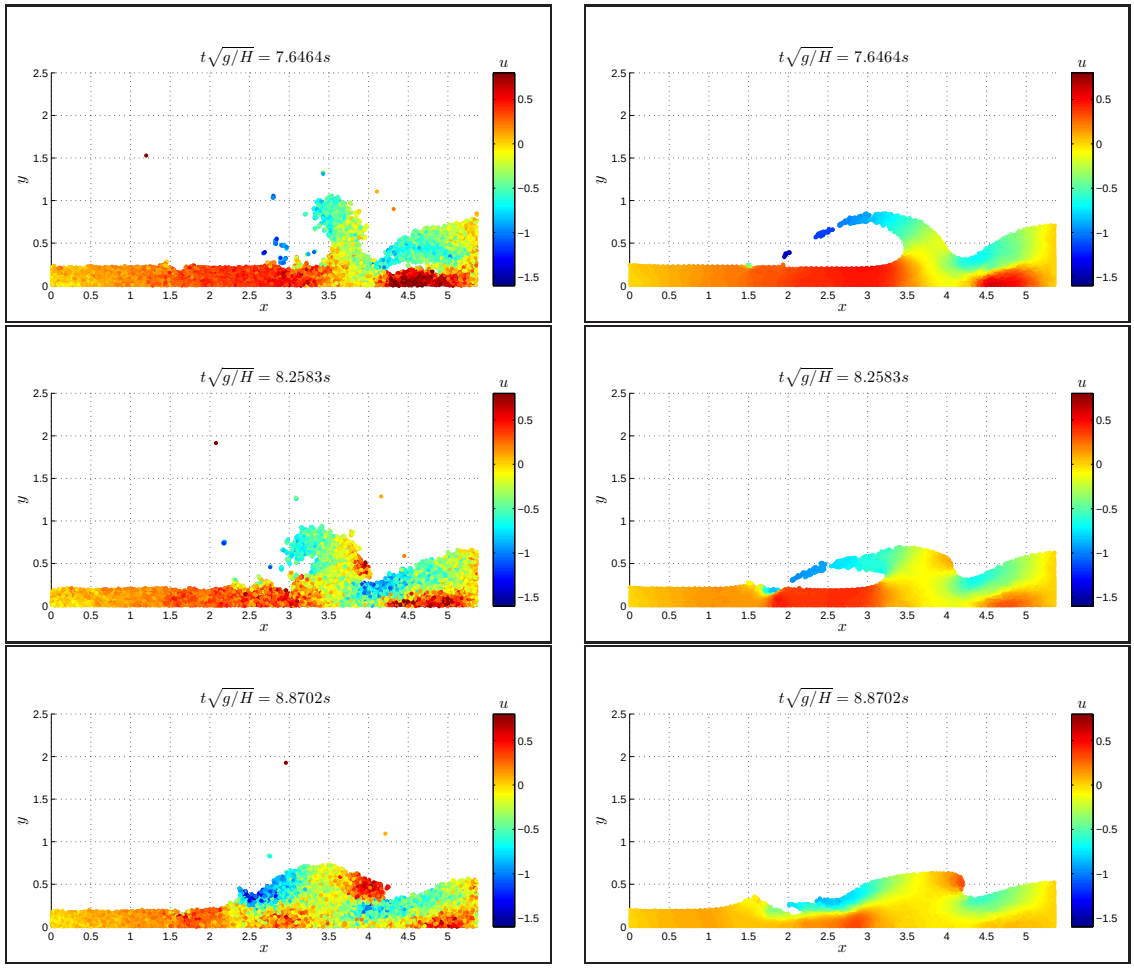

Figura 5.6: Velocidade horizontal (a) sem suavização e (b) com suavização aplicada em n = 5 passos, em $t = 7.6464s$, $t = 8.2583s$ e $t = 8.8702s$.

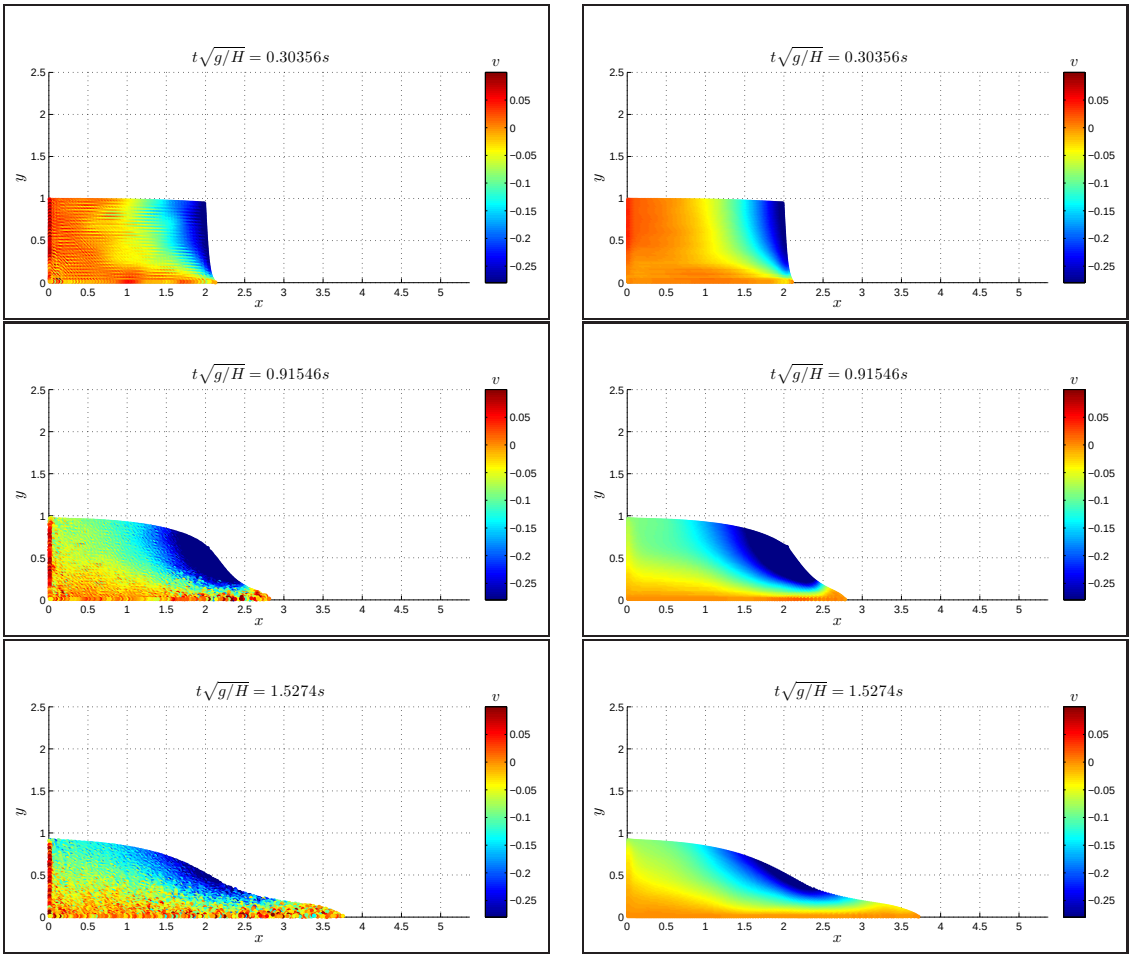

Figura 5.7: Velocidade vertical (a) sem suavização e (b) com suavização aplicada em n = 5 passos, em $t = 0.30356s$, $t = 0.91546s$ e $t = 1.5274s$.

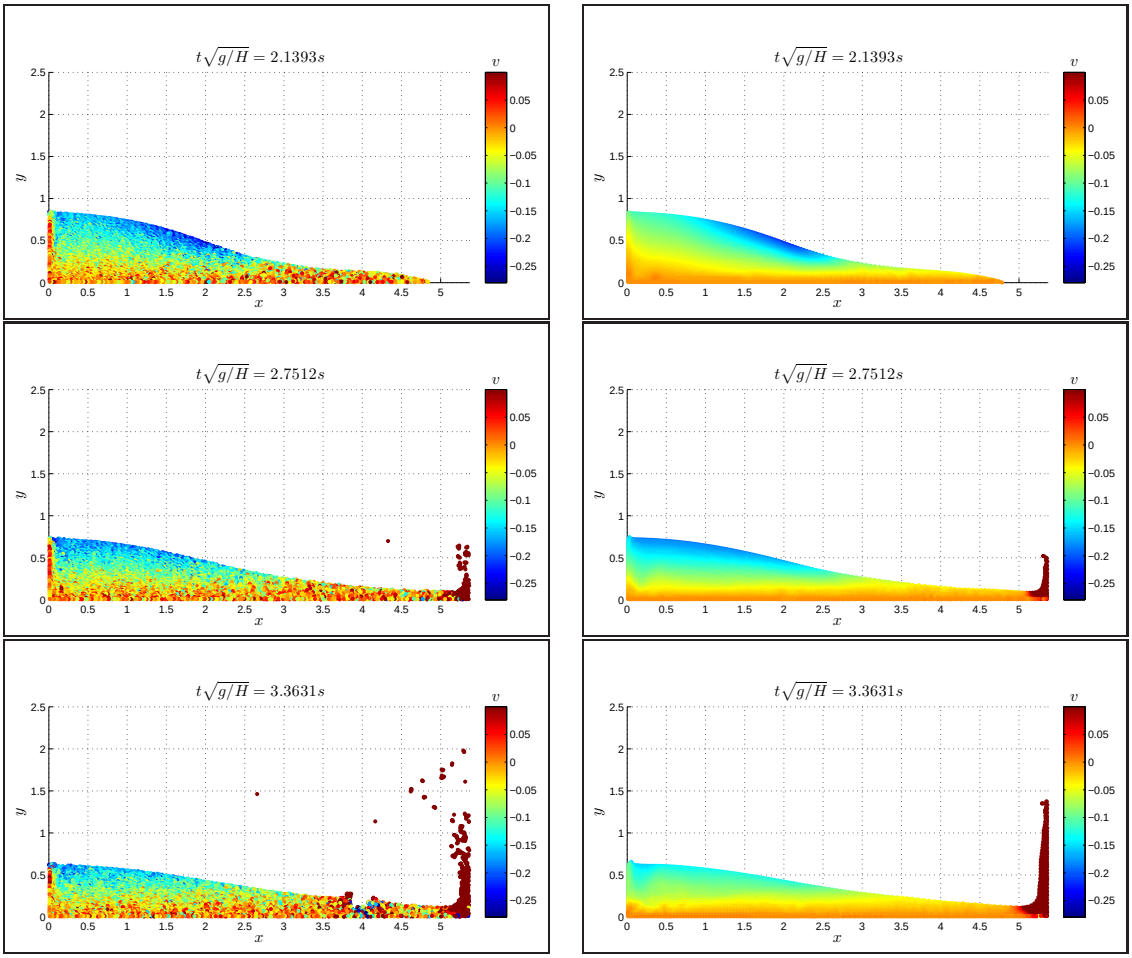

Figura 5.8: Velocidade vertical (a) sem suavização e (b) com suavização aplicada em n = 5 passos, em $t = 2.1393s$, $t = 2.7512s$ e $t = 3.3631s$.

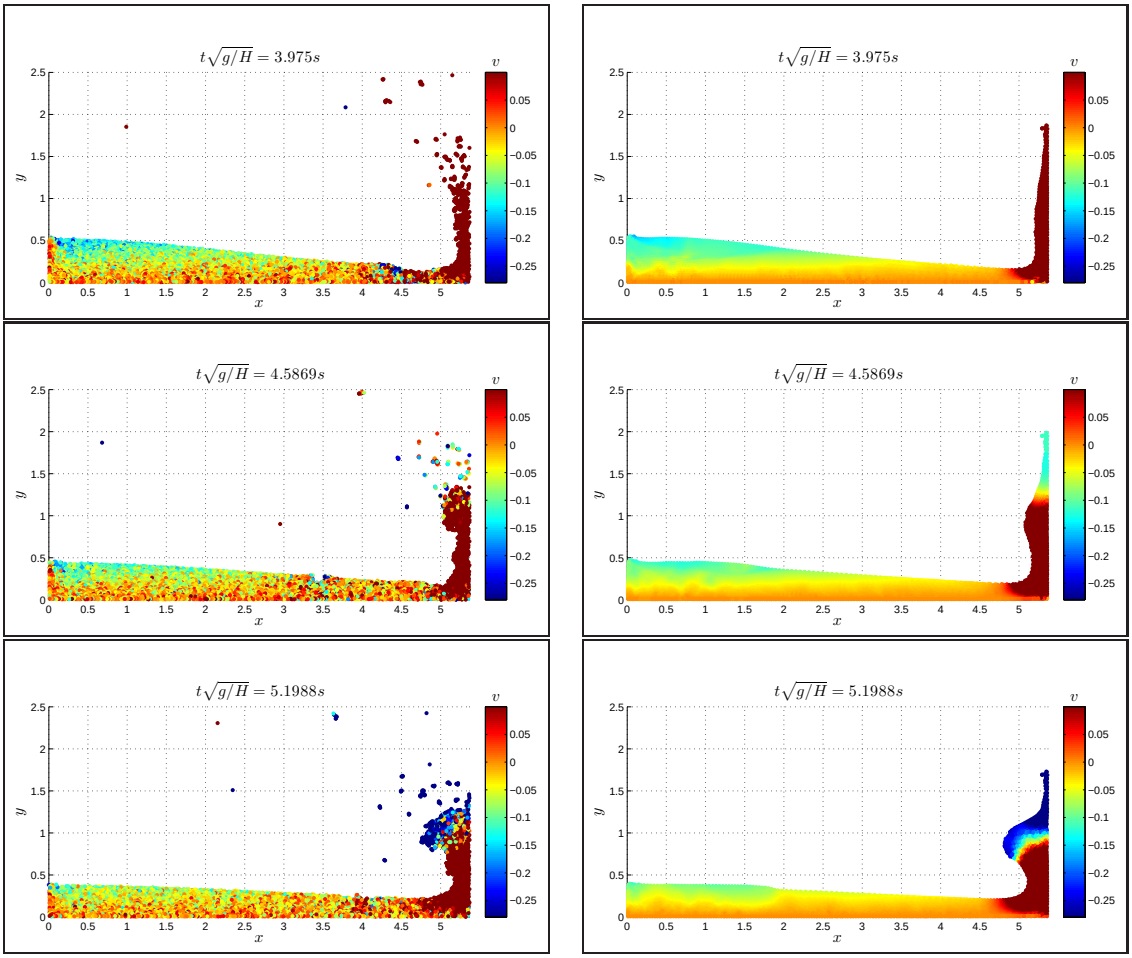

Figura 5.9: Velocidade vertical (a) sem suavização e (b) com suavização aplicada em n = 5 passos, em $t = 3.975s$, $t = 4.5869s$ e $t = 5.1988s$.

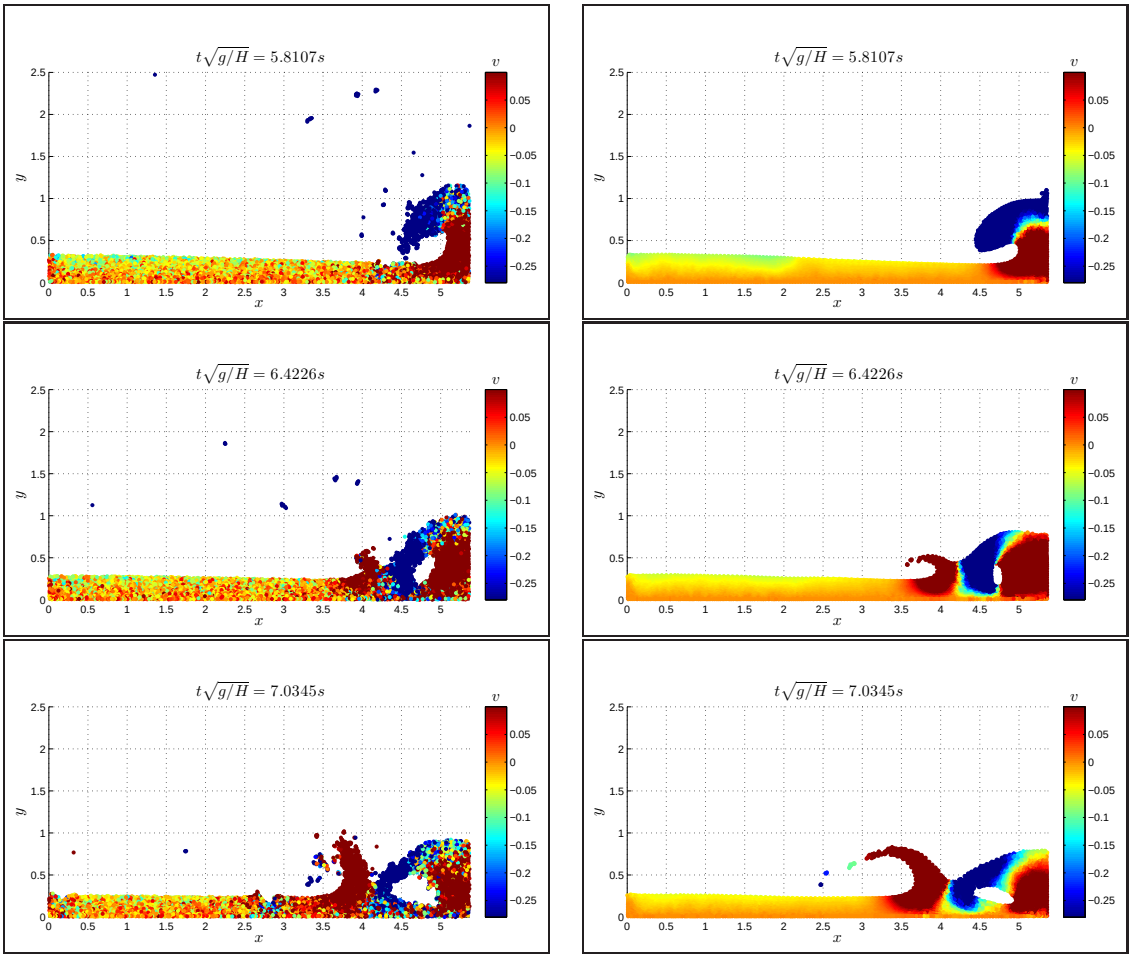

Figura 5.10: Velocidade vertical (a) sem suavização e (b) com suavização aplicada em n = 5 passos, em $t = 5.8107s$, $t = 6.4226s$ e $t = 7.0345s$.

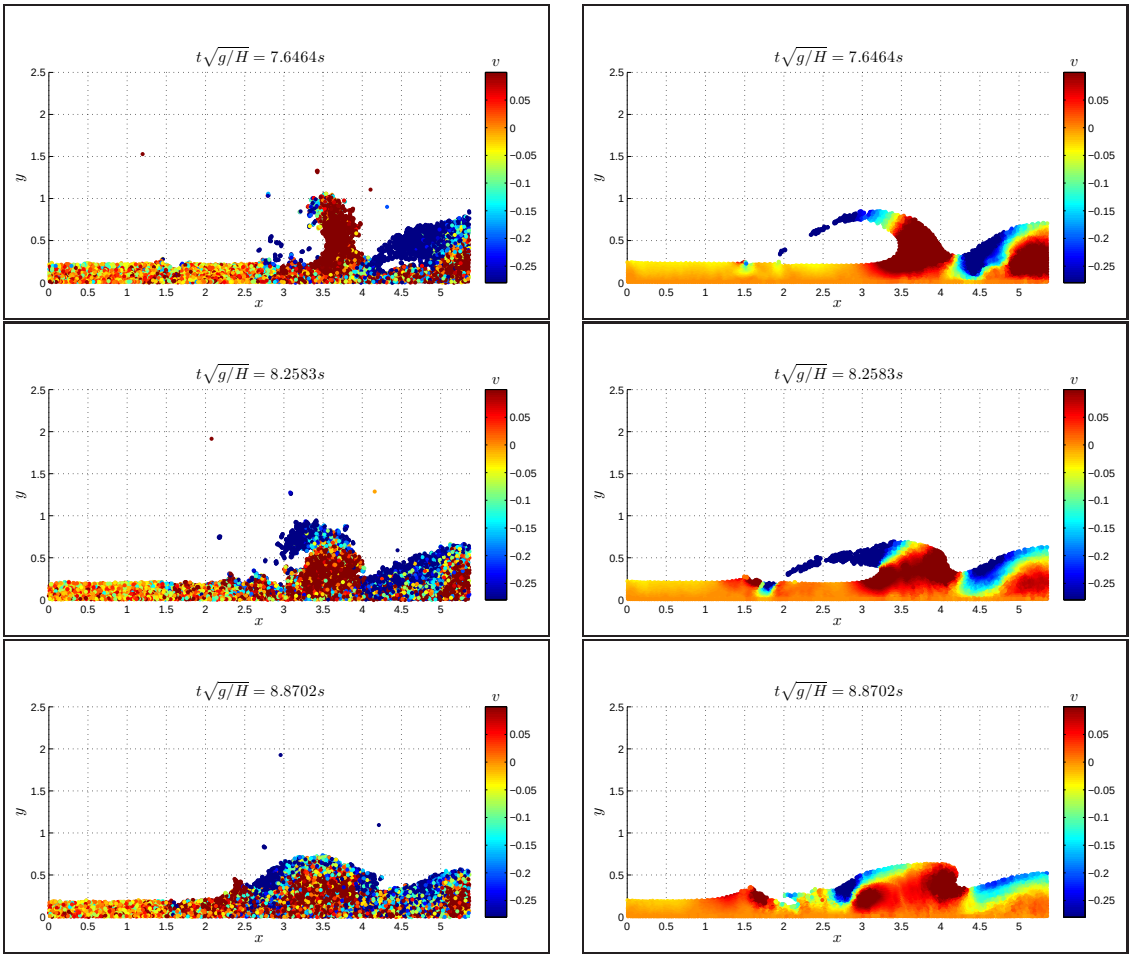

Figura 5.11: Velocidade vertical (a) sem suavização e (b) com suavização aplicada em n = 5 passos, em $t = 7.6464s$, $t = 8.2583s$ e $t = 8.8702s$.

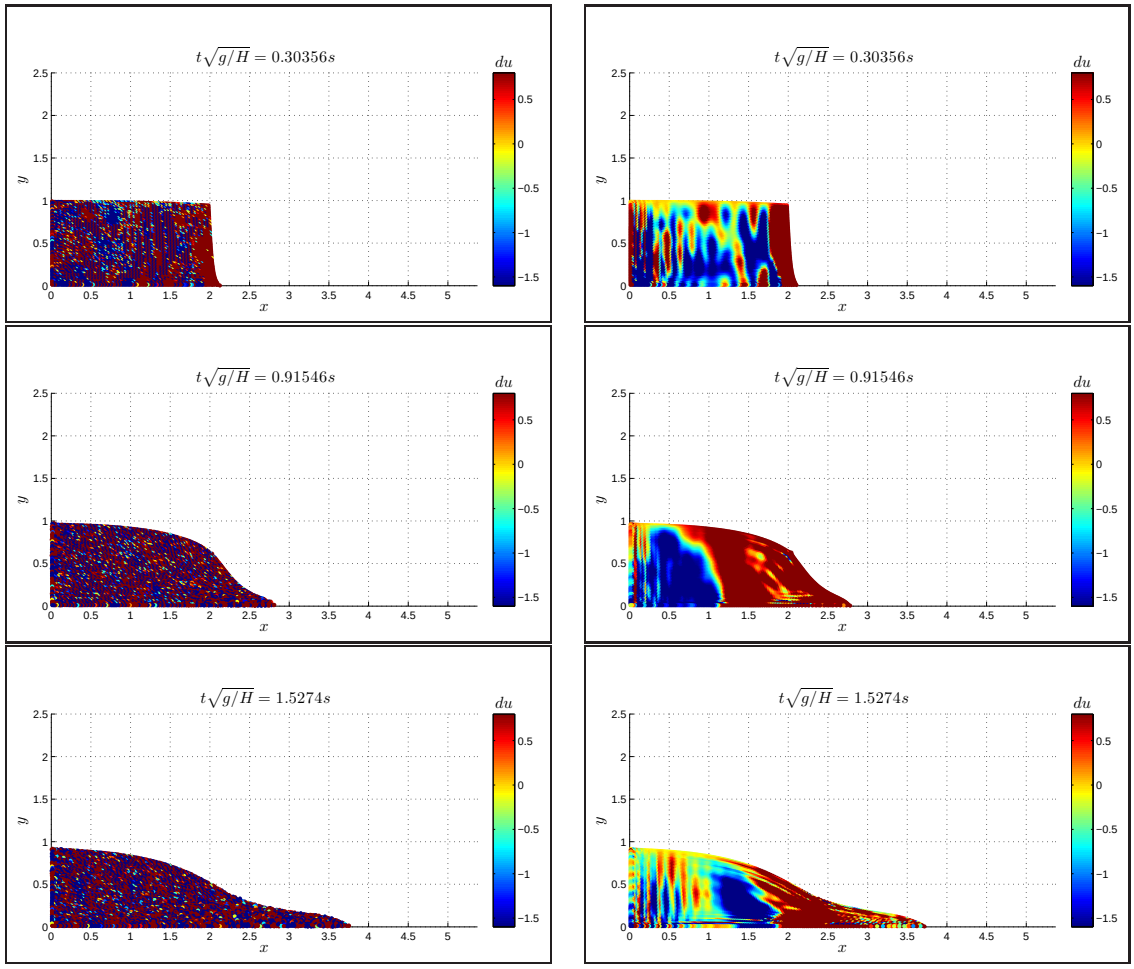

Figura 5.12: Aceleração horizontal (a) sem suavização e (b) com suavização aplicada em n = 5 passos, em $t = 0.30356s$, $t = 0.91546s$ e $t = 1.5274s$.

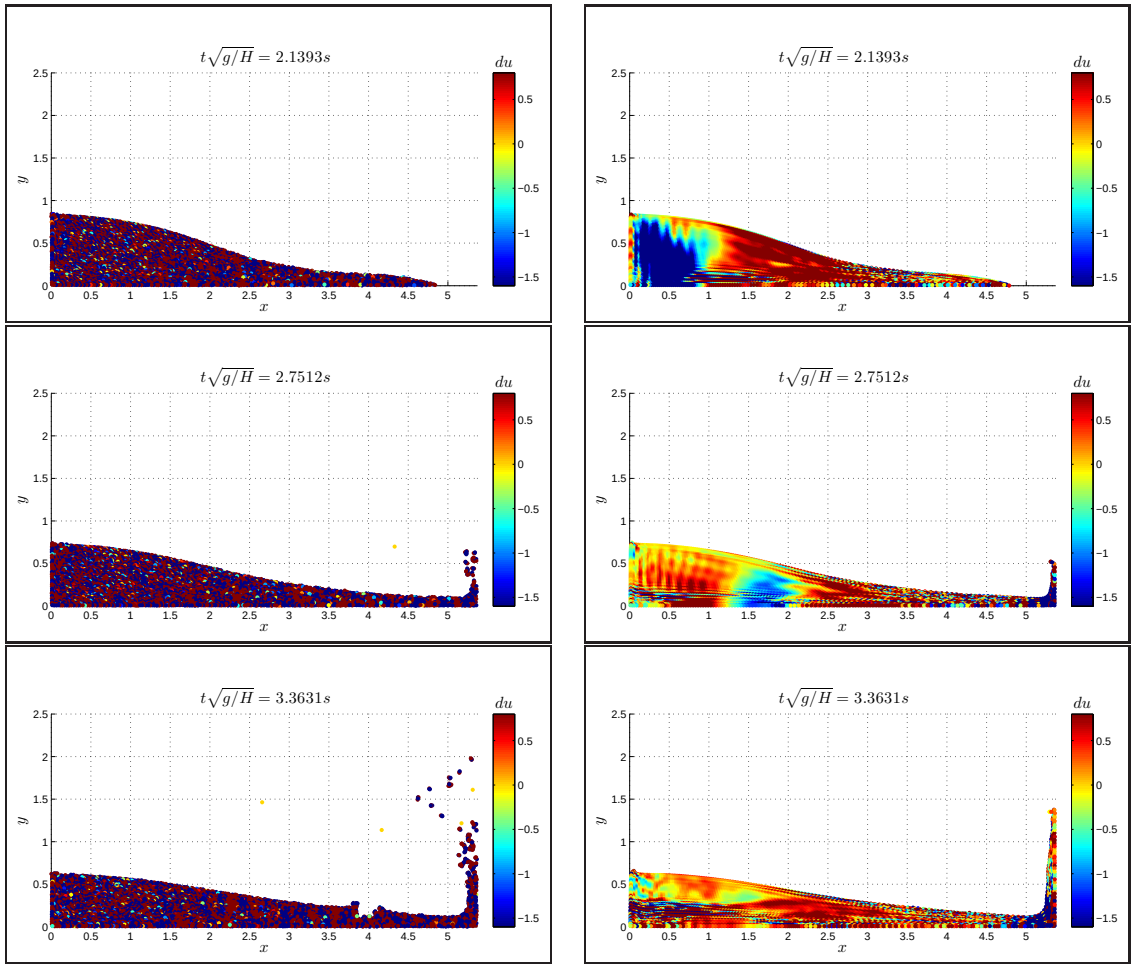

Figura 5.13: Aceleração horizontal (a) sem suavização e (b) com suavização aplicada em n = 5 passos, em $t = 2.1393s$, $t = 2.7512s$ e $t = 3.3631s$.

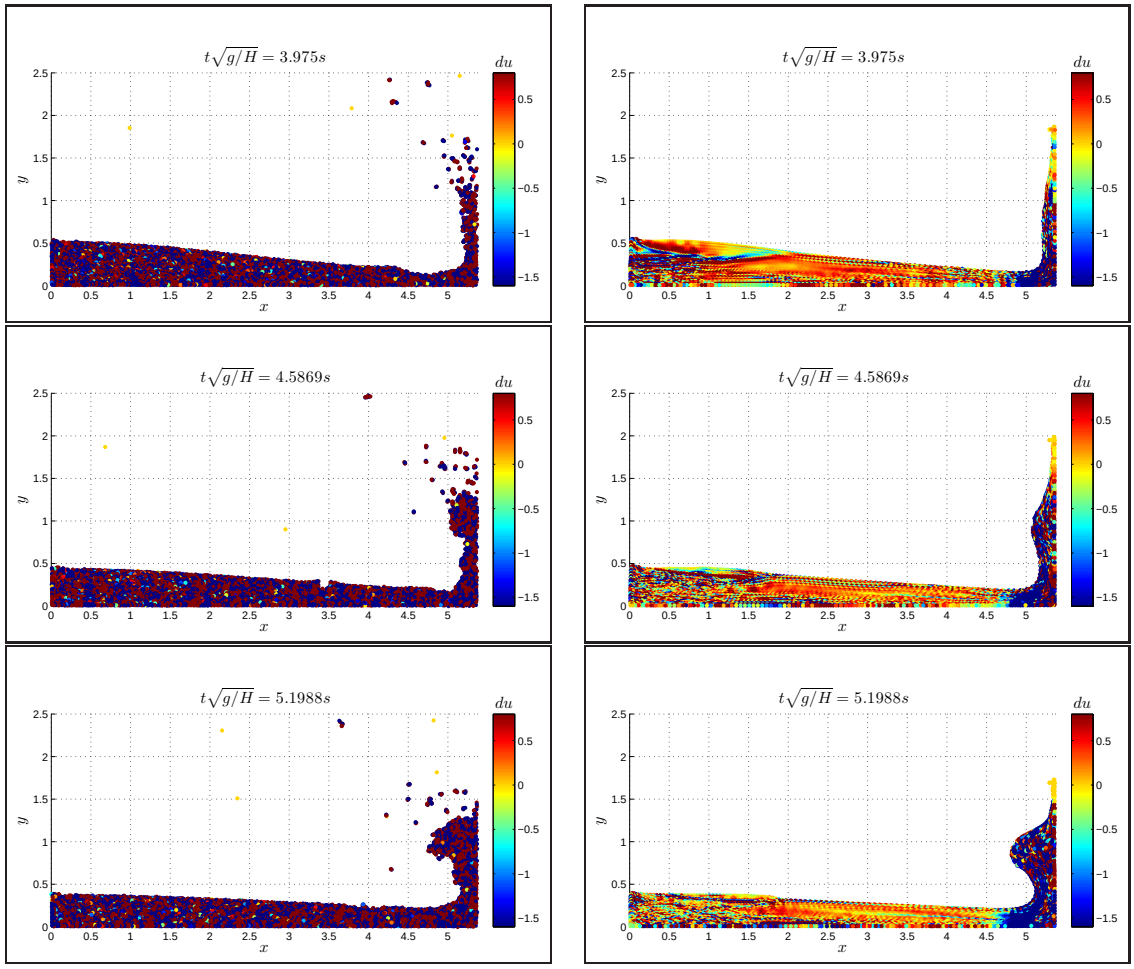

Figura 5.14: Aceleração horizontal (a) sem suavização e (b) com suavização aplicada em n = 5 passos, em $t = 3.975s$, $t = 4.5869s$ e $t = 5.1988s$.

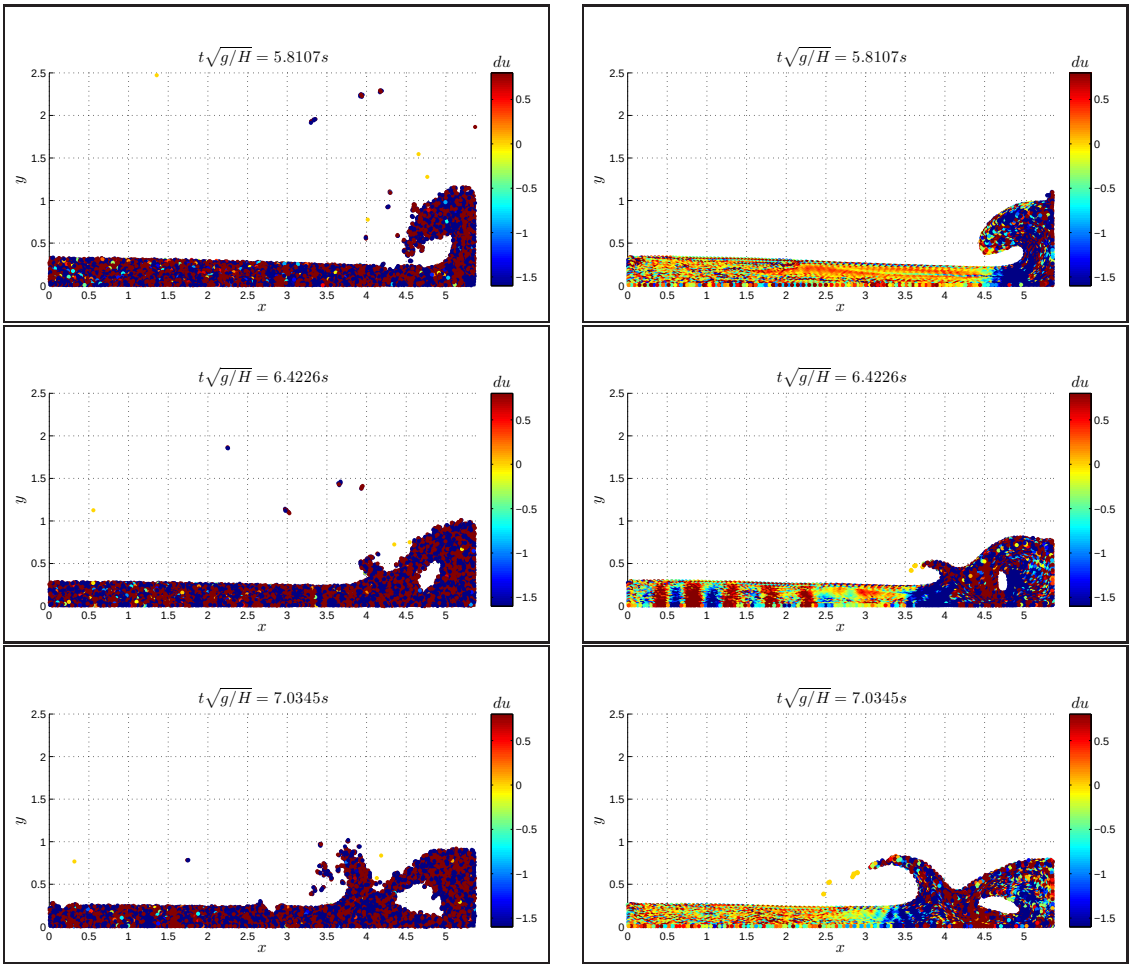

Figura 5.15: Aceleração horizontal (a) sem suavização e (b) com suavização aplicada em n = 5 passos, em $t = 5.8107s$, $t = 6.4226s$ e $t = 7.0345s$.

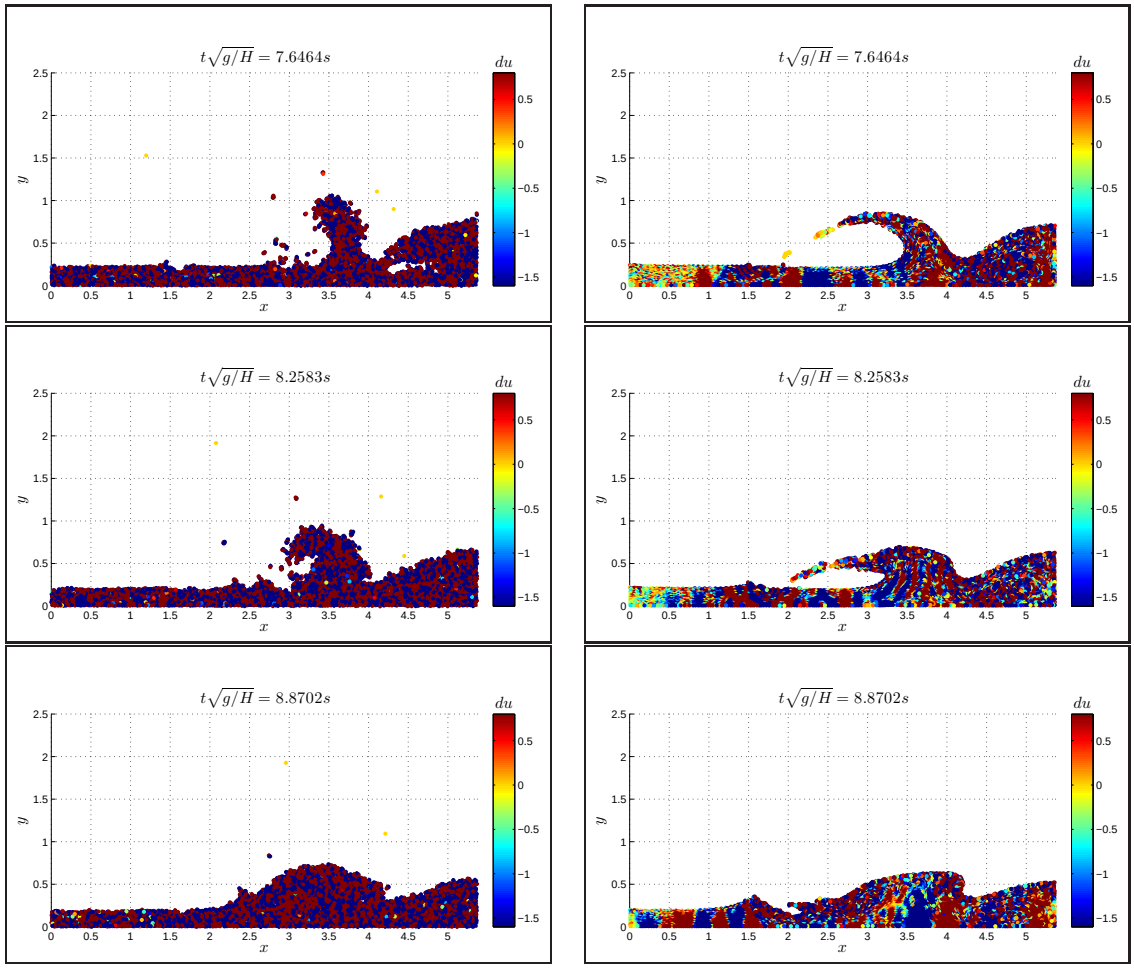

Figura 5.16: Aceleração horizontal (a) sem suavização e (b) com suavização aplicada em n = 5 passos, em $t = 7.6464s$, $t = 8.2583s$ e $t = 8.8702s$.

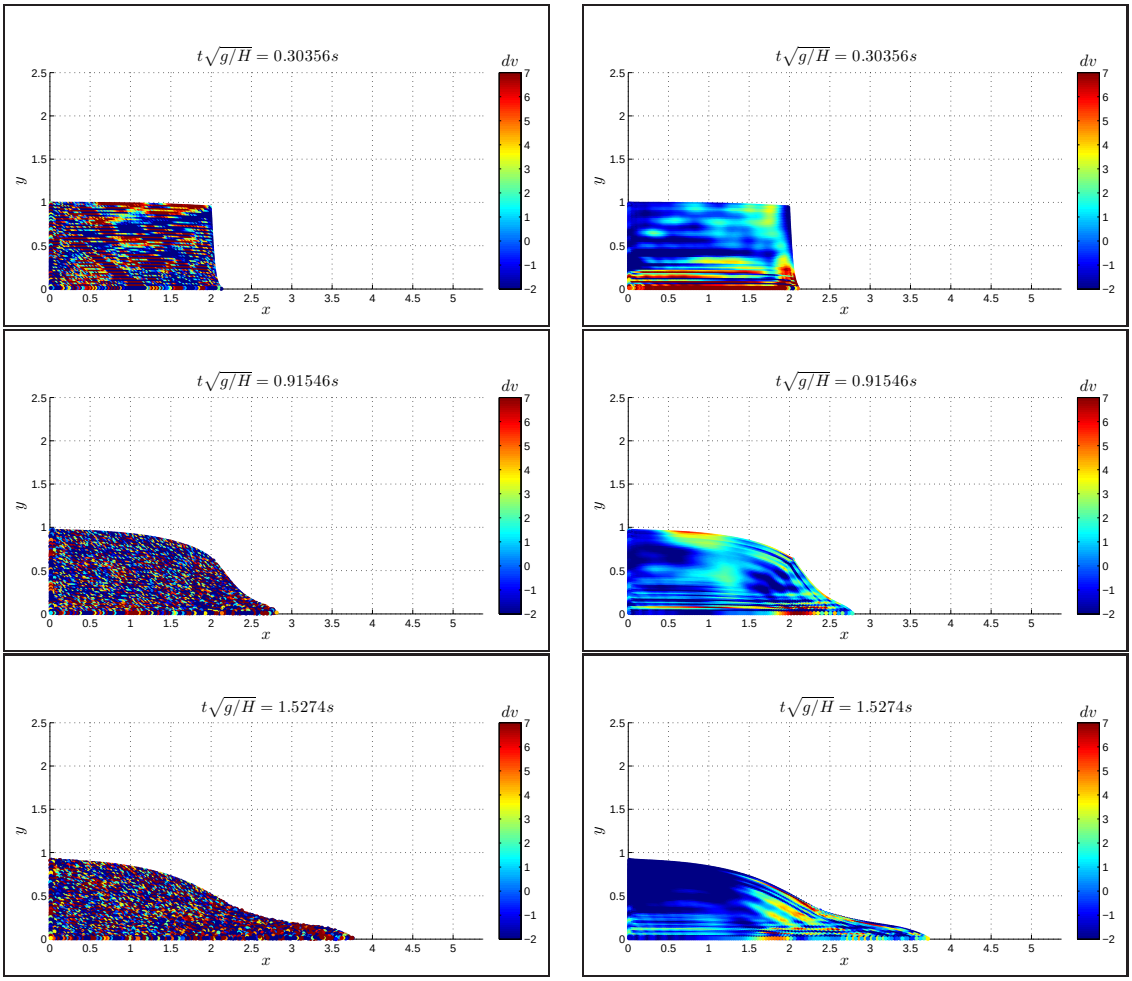

Figura 5.17: Aceleração vertical (a) sem suavização e (b) com suavização aplicada em n = 5 passos, em $t = 0.30356s$, $t = 0.91546s$ e $t = 1.5274s$.

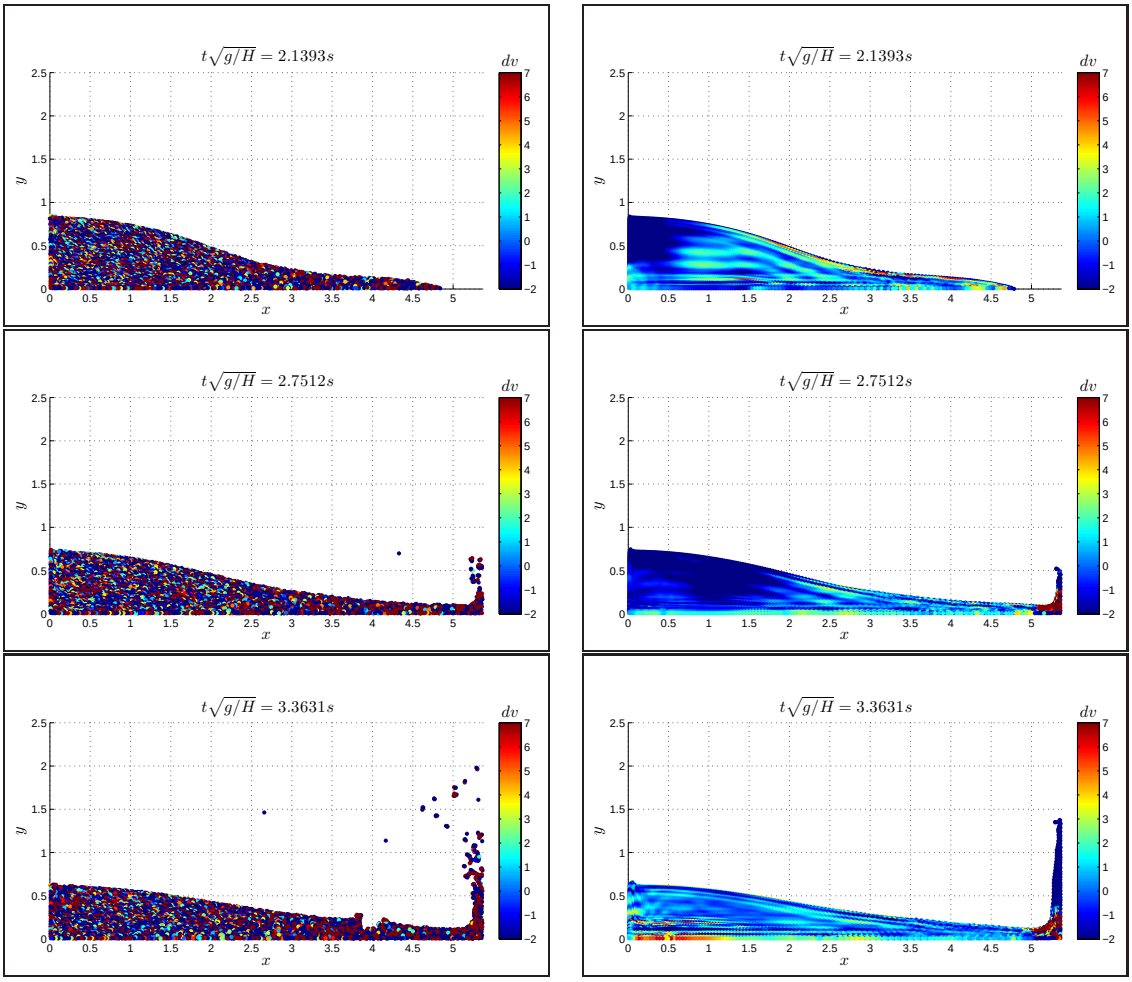

Figura 5.18: Aceleração vertical (a) sem suavização e (b) com suavização aplicada em n = 5 passos, em $t = 2.1393s$, $t = 2.7512s$ e $t = 3.3631s$.

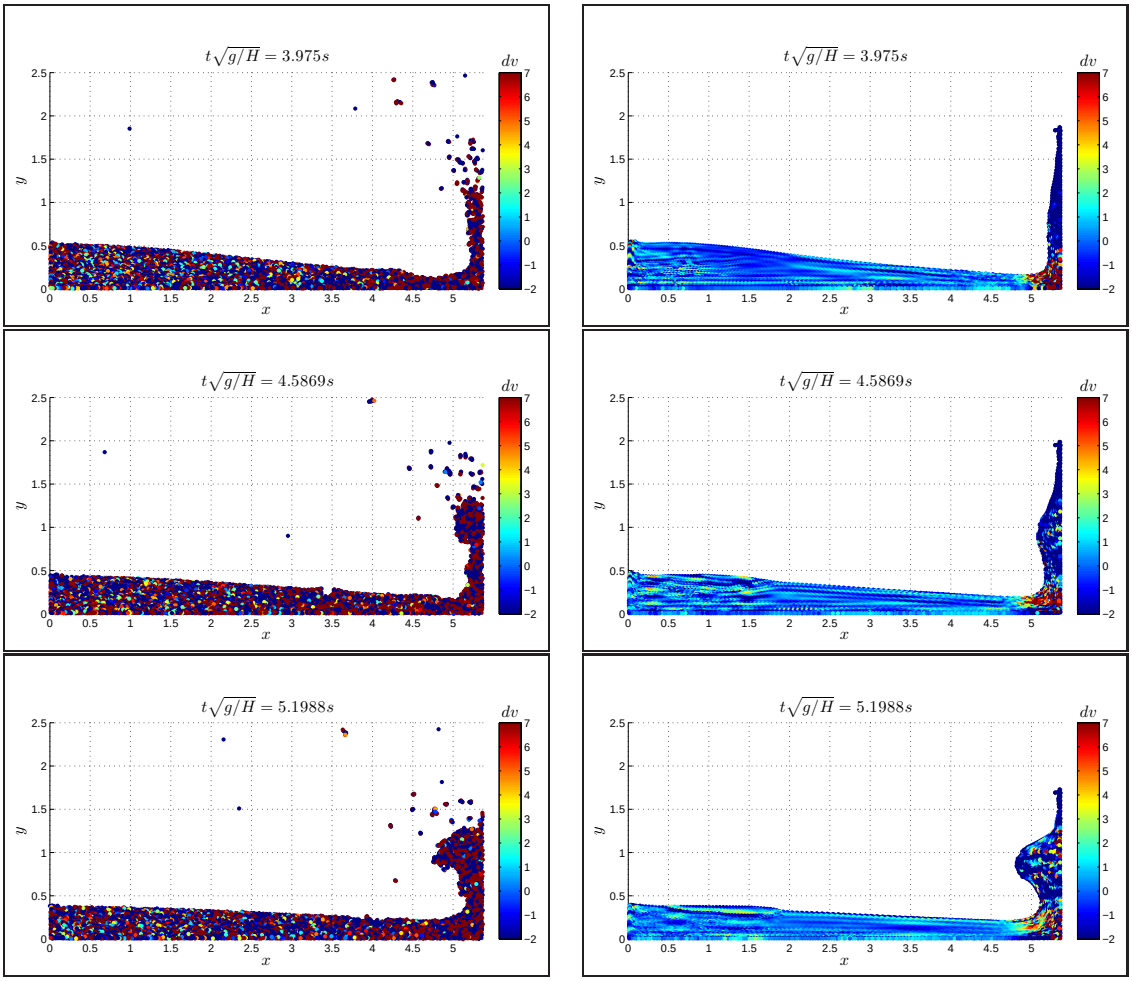

Figura 5.19: Aceleração vertical (a) sem suavização e (b) com suavização aplicada em n = 5 passos, em $t = 3.975s$, $t = 4.5869s$ e $t = 5.1988s$.

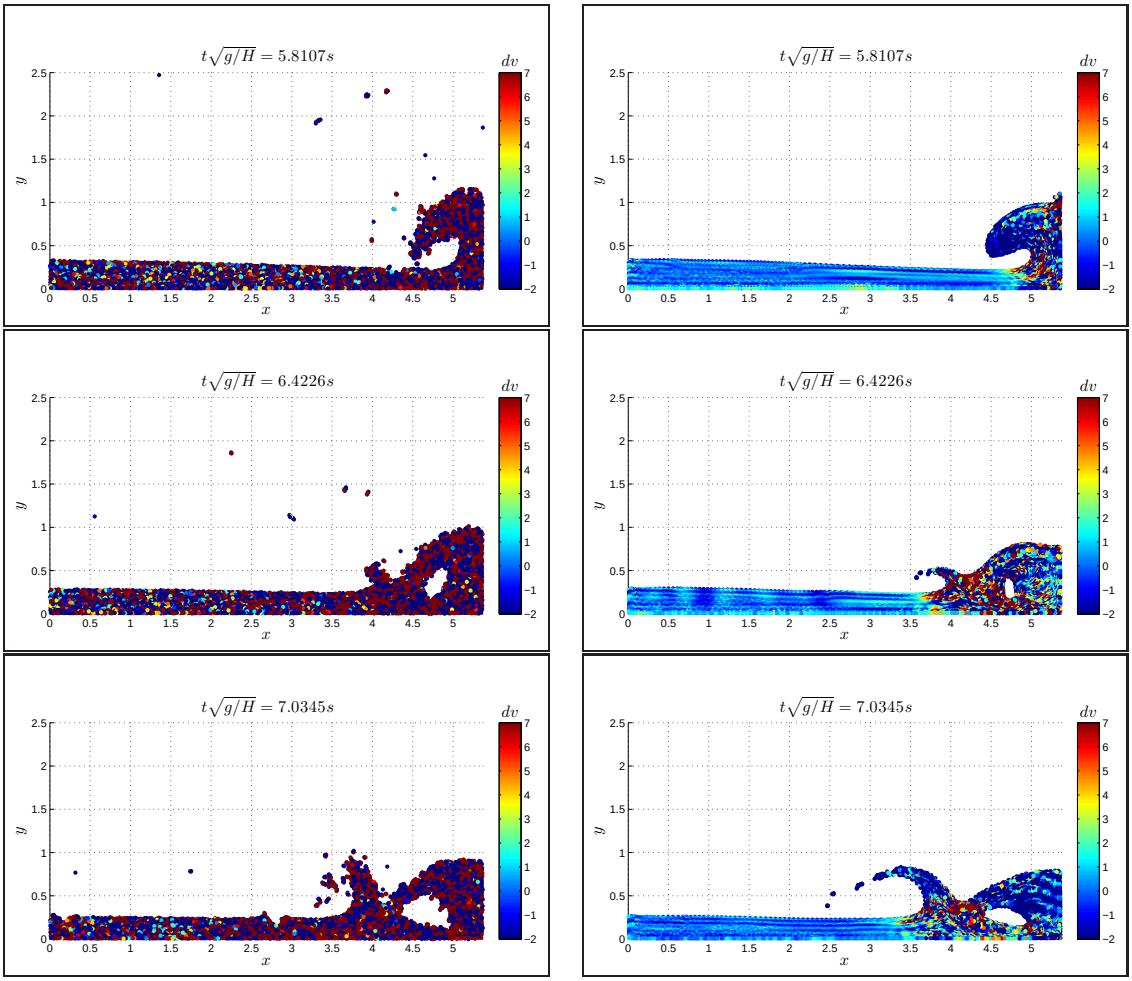

Figura 5.20: Aceleração vertical (a) sem suavização e (b) com suavização aplicada em n = 5 passos, em $t = 5.8107s$, $t = 6.4226s$ e $t = 7.0345s$.

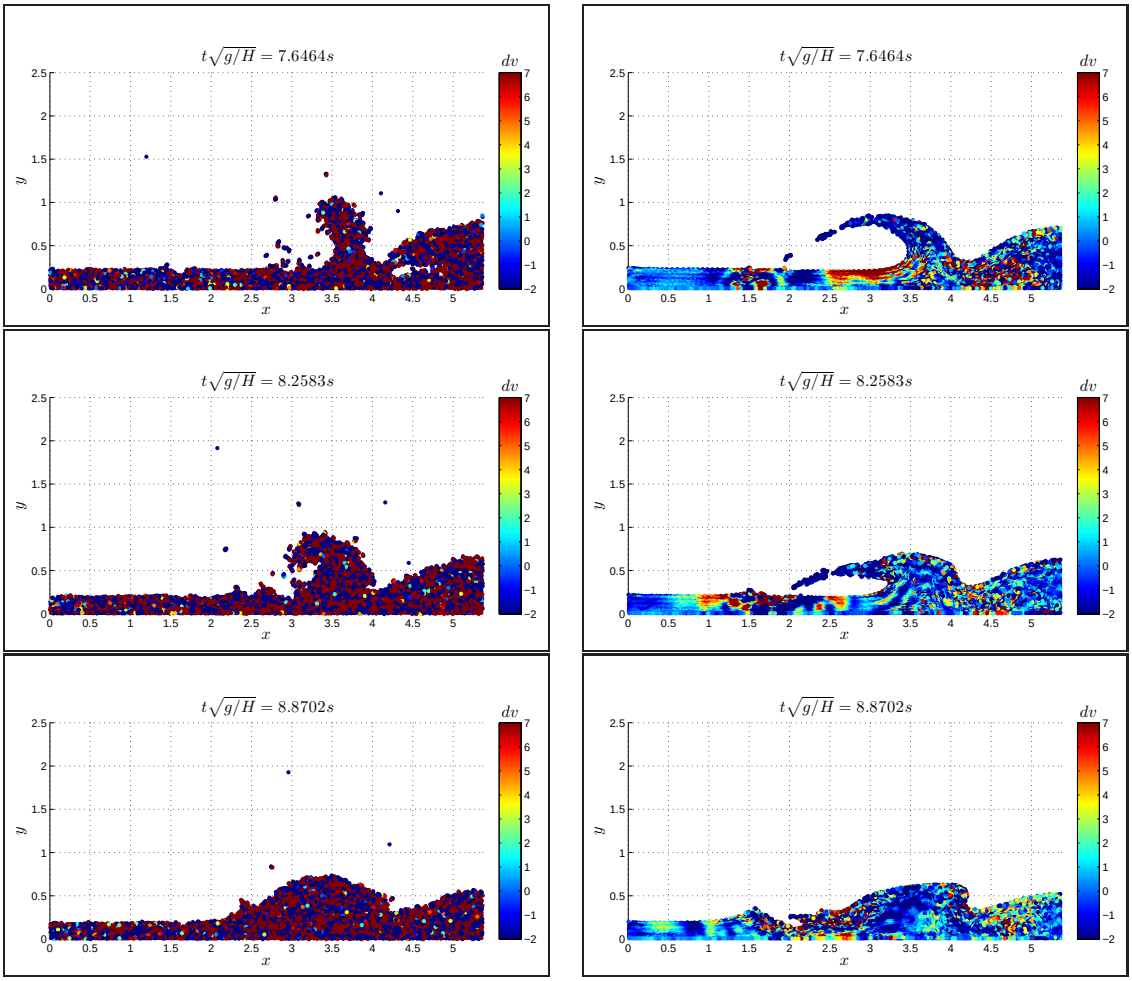

Figura 5.21: Aceleração vertical (a) sem suavização e (b) com suavização aplicada em n = 5 passos, em $t = 7.6464s$, $t = 8.2583s$ e $t = 8.8702s$.

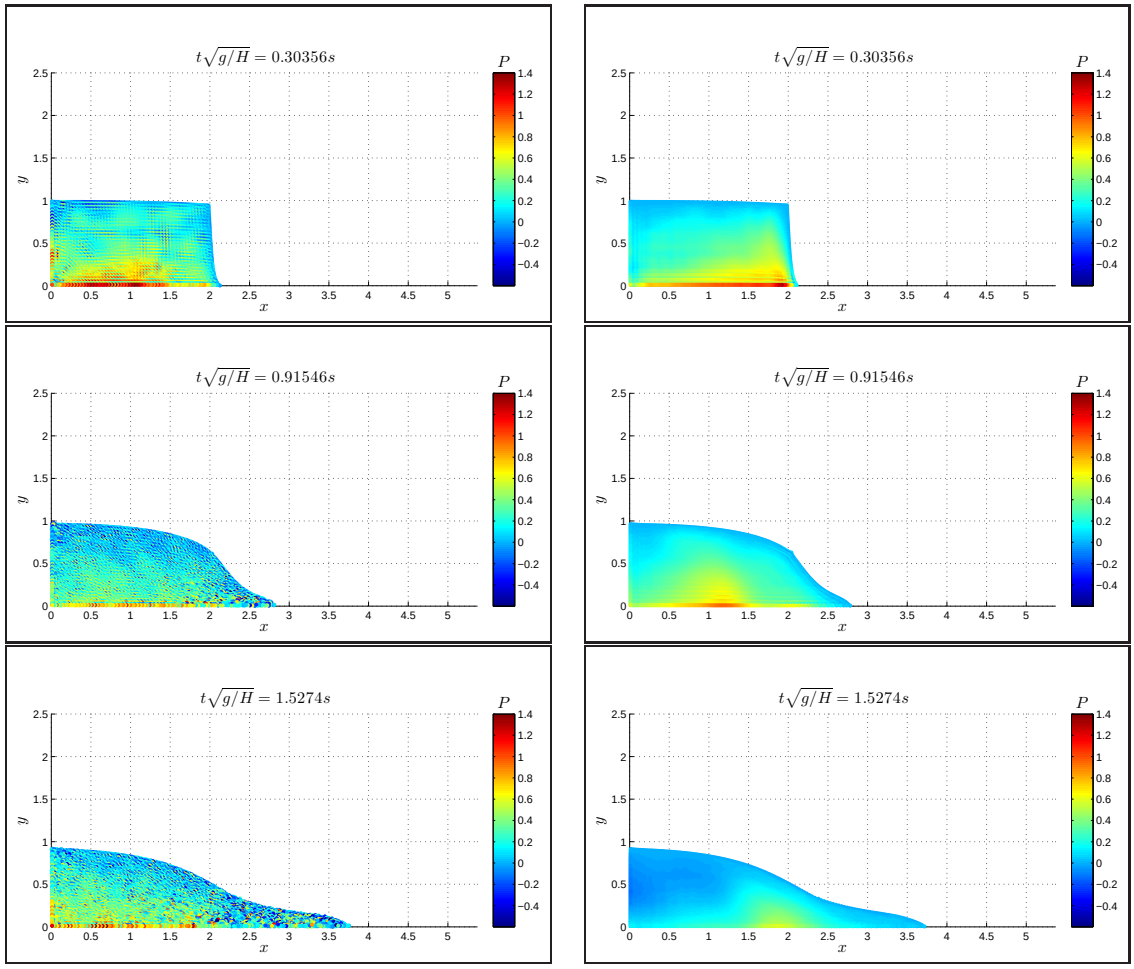

Figura 5.22: Pressão (a) sem suavização e (b) com suavização aplicada em n = 5 passos, em $t = 0.30356s$, $t = 0.91546s$ e $t = 1.5274s$.

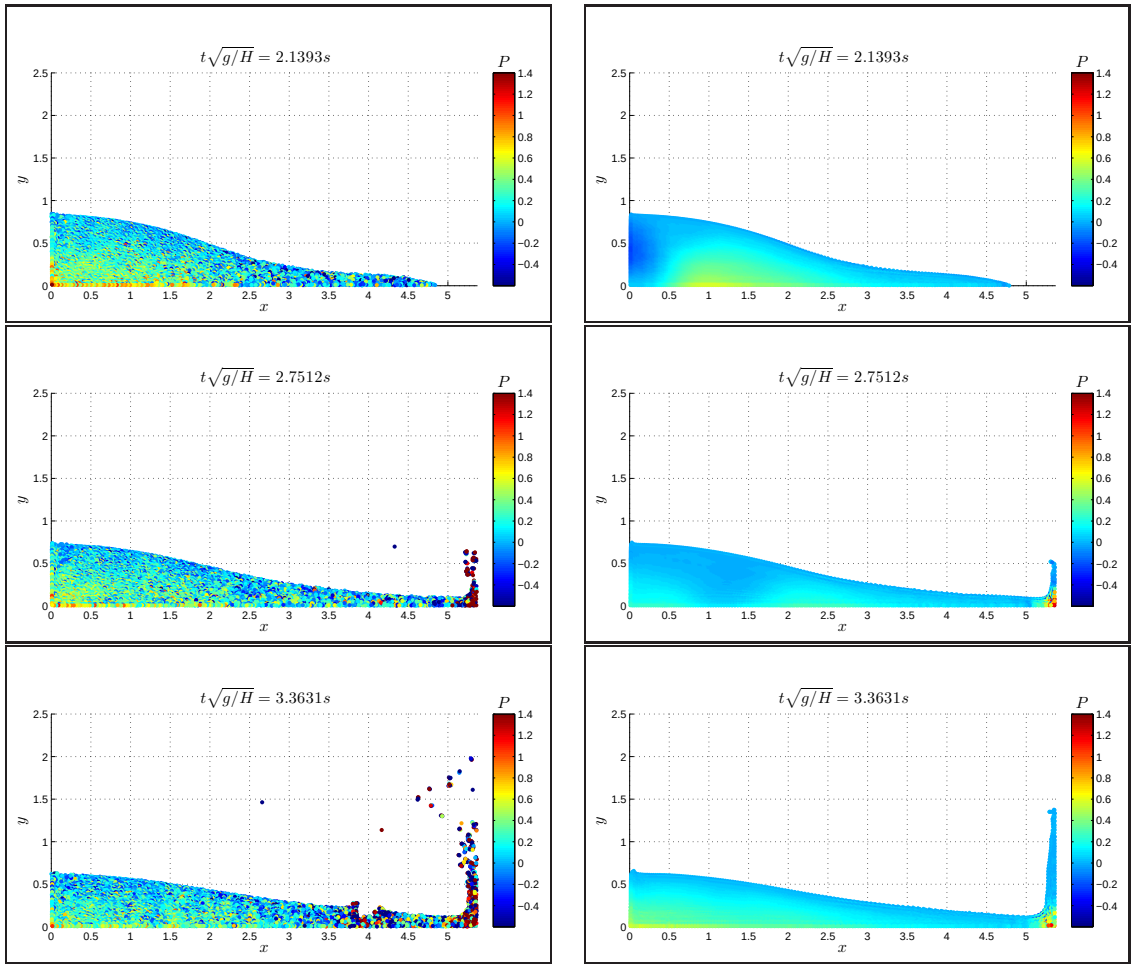

Figura 5.23: Pressão (a) sem suavização e (b) com suavização aplicada em n = 5 passos, em $t = 2.1393s$, $t = 2.7512s$ e $t = 3.3631s$.

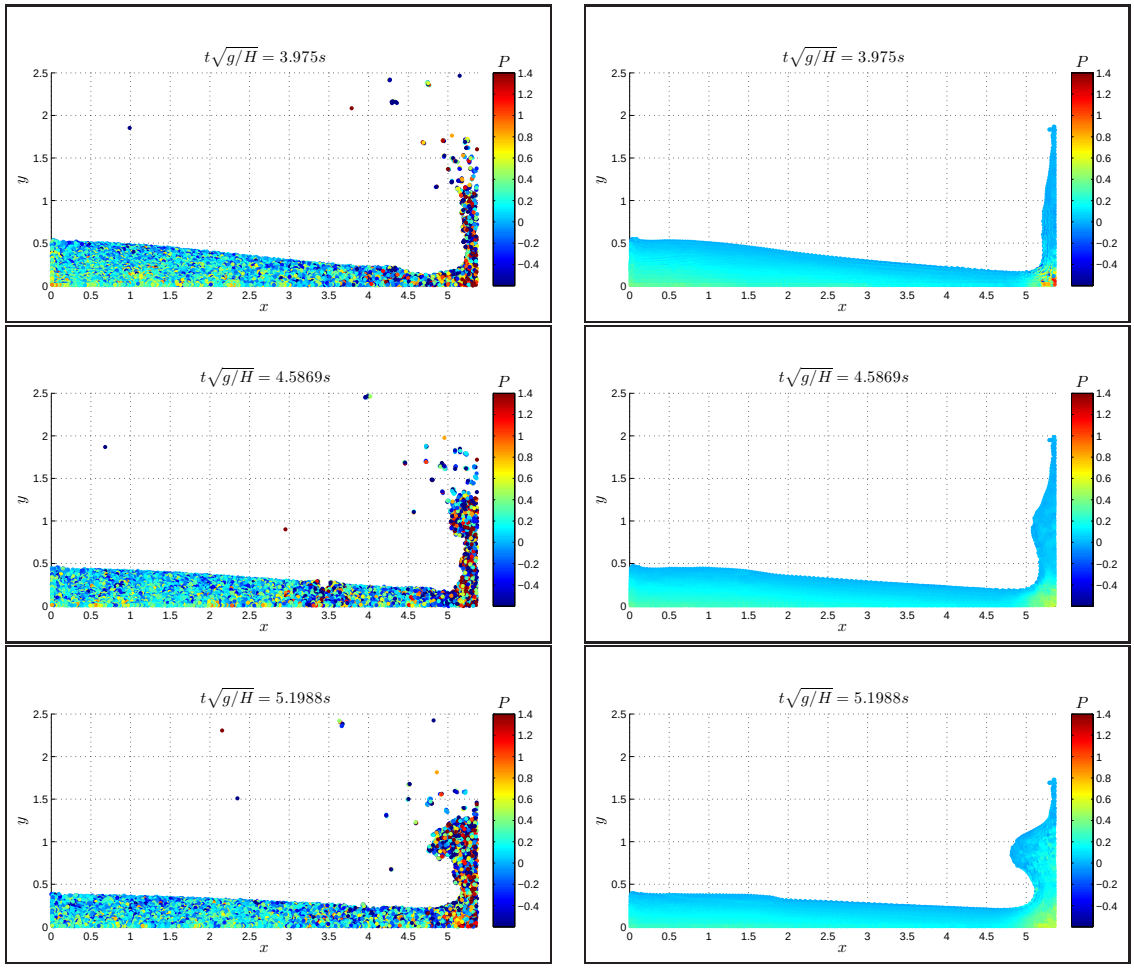

Figura 5.24: Pressão (a) sem suavização e (b) com suavização aplicada em n = 5 passos, em $t = 3.975s$, $t = 4.5869s$ e $t = 5.1988s$.

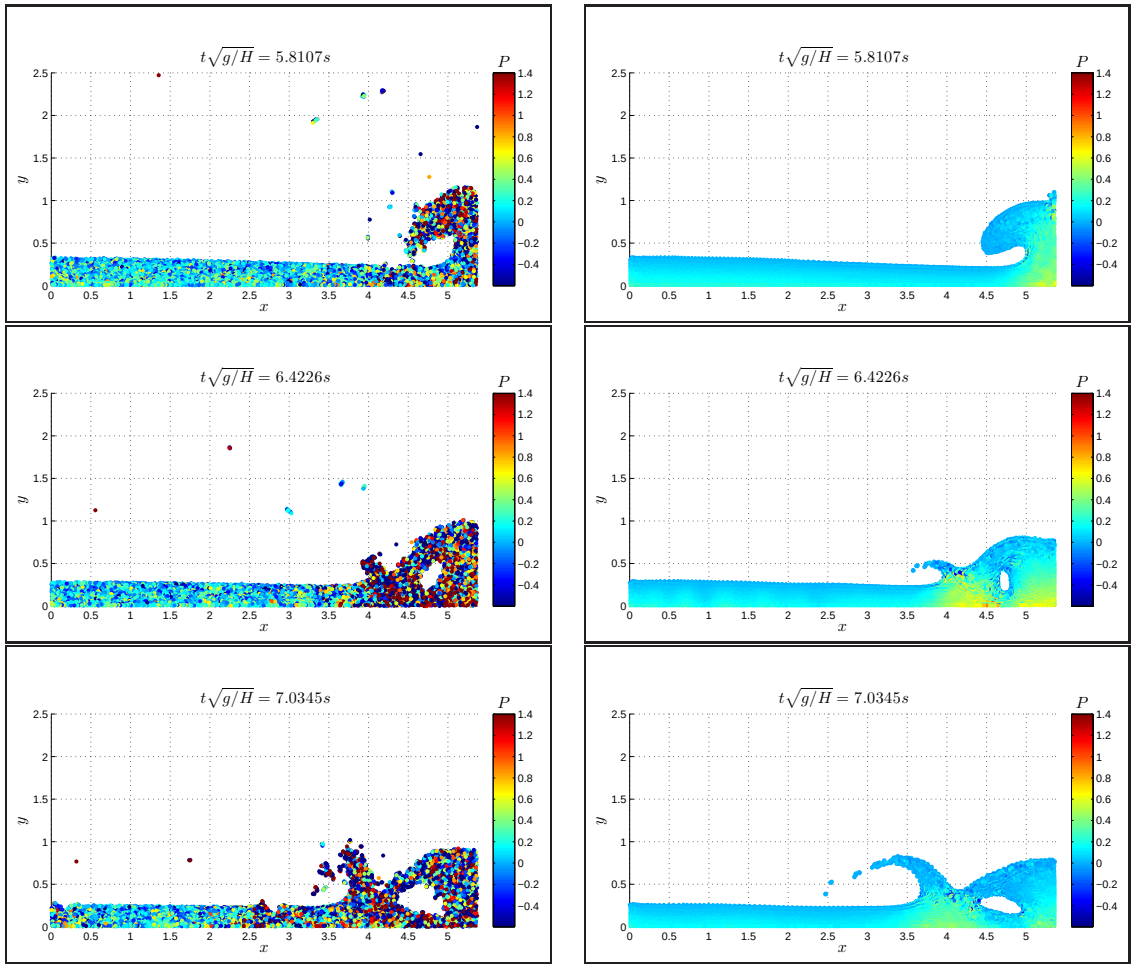

Figura 5.25: Pressão (a) sem suavização e (b) com suavização aplicada em n = 5 passos, em $t = 5.8107s$, $t = 6.4226s$ e $t = 7.0345s$.

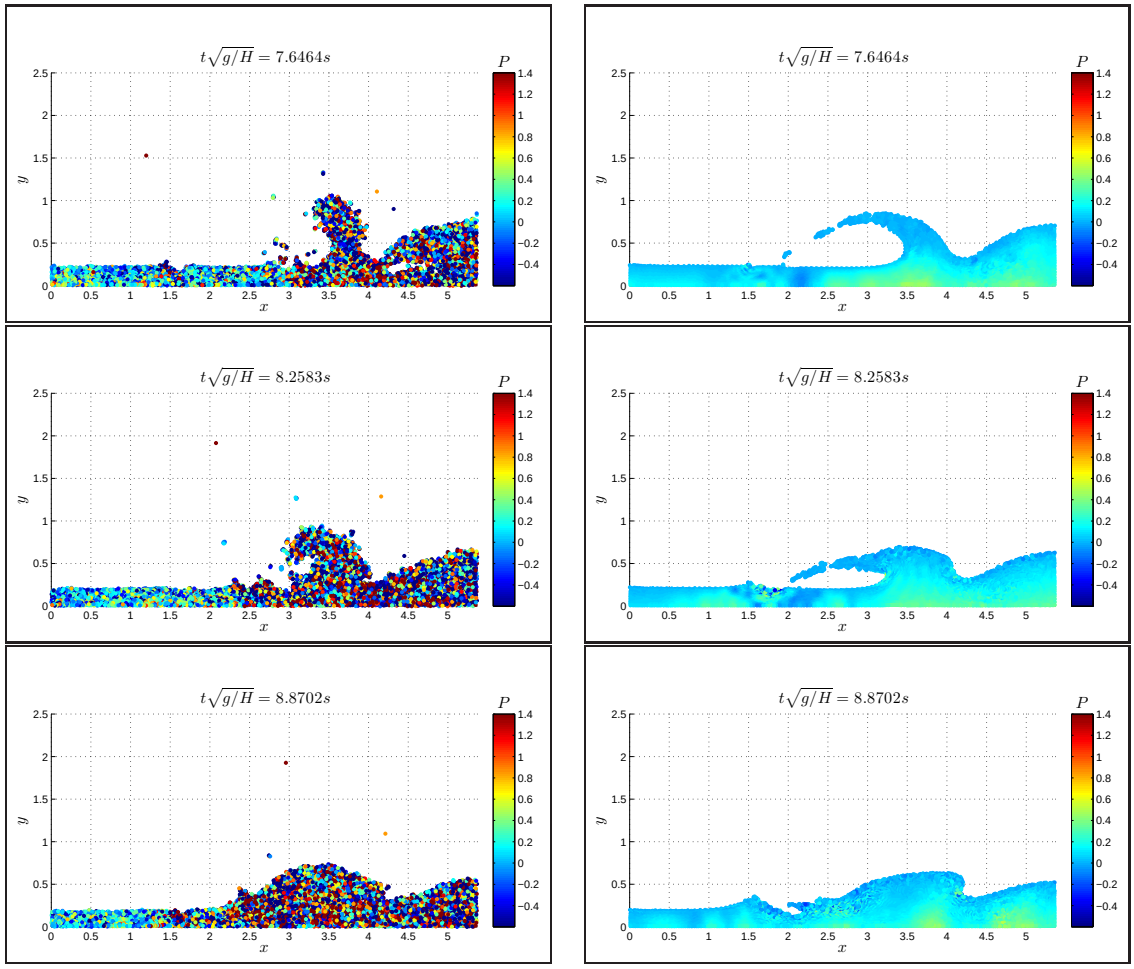

Figura 5.26: Pressão (a) sem suavização e (b) com suavização aplicada em n = 5 passos, em $t = 7.6464s$, $t = 8.2583s$ e $t = 8.8702s$.

A fig. 5.27 exibe a evolução da frente do fluido, *i.e.*, a coordenada mais a direita x_f, mostrando que os resultados estão de acordo com outros obtidos na literatura [6]. As alturas do fluido nos pontos $A = (3,721H, 0)$ e $B = (4,542H, 0)$ (ver fig. 5.28 e 5.29) foram extraídos e comparados com [6] e [58]. A evolução da pressão de impacto na parede, i.e., no ponto C, exibida na fig. 5.30. Os valores obtidos para a pressão no ponto C são comparados com as simulações de [58] e [6]. Vale lembrar que a modelagem apresentada nesta dissertação não leva em conta qualquer influência do ar.

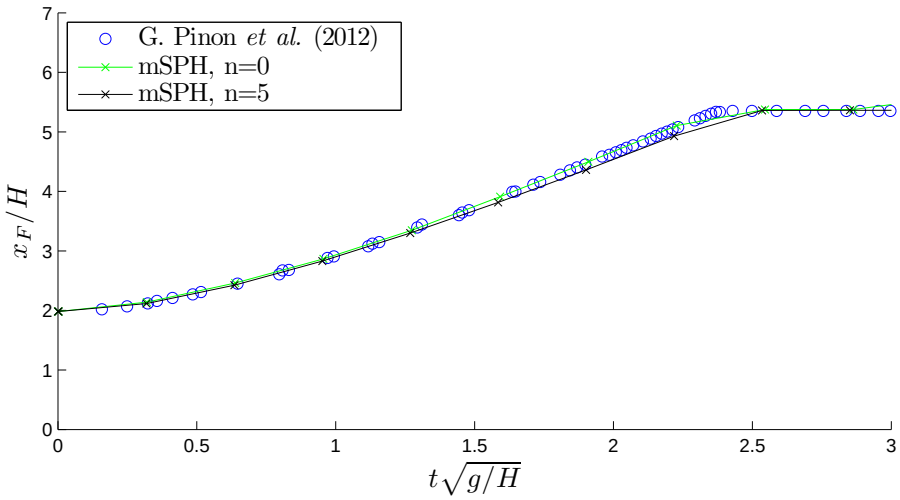

Figura 5.27: Evolução da frente da onda após a quebra da barragem (ver a fig. 5.1). As simulações mSPH são comparadas com os resultados numéricos de [6].

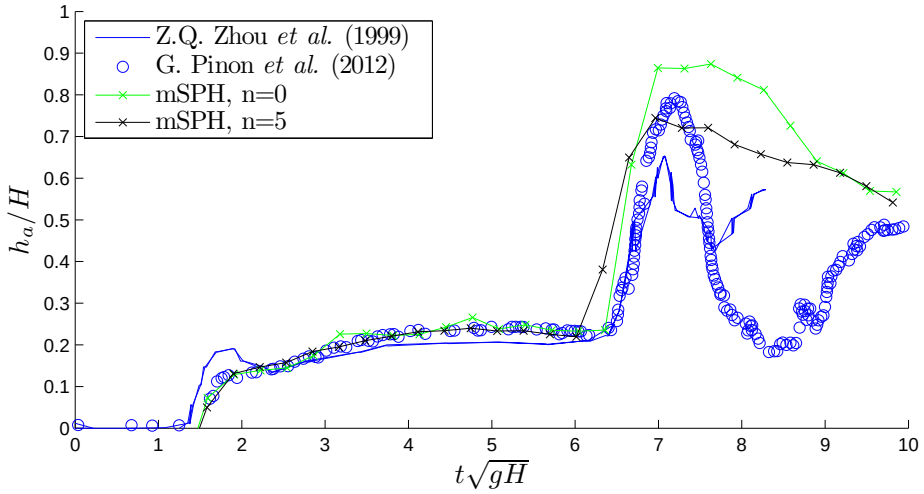

Figura 5.28: Evolução da altura máxima do fluido no ponto A (ver a fig. 5.1). As simulações mSPH são comparadas com o resultado numérico de [6] e os dados experimentais de [58].

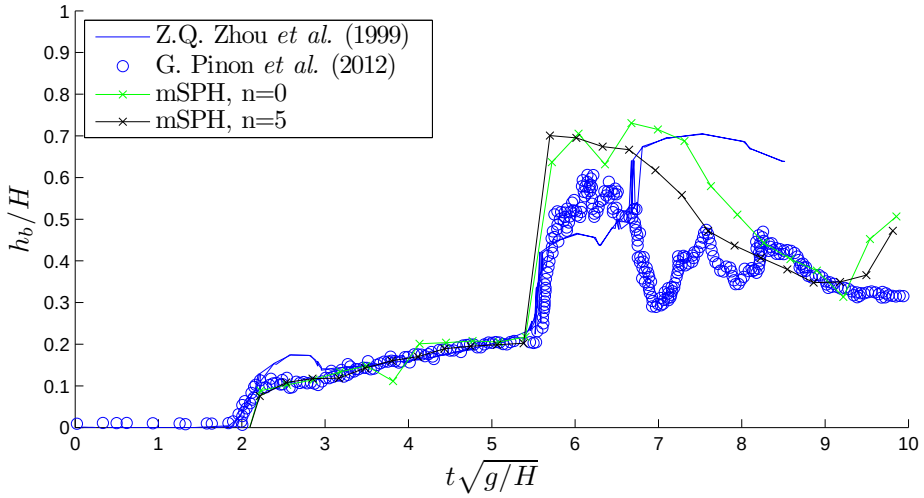

Figura 5.29: Evolução da altura máxima do fluido no ponto B (ver a fig. 5.1). As simulações mSPH são comparadas com o resultado numérico de [6] e os dados experimentais de [58].

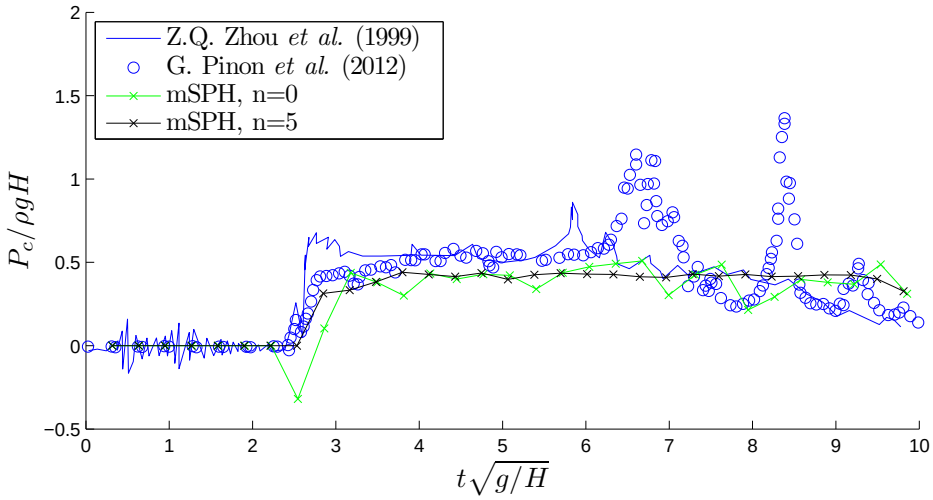

Figura 5.30: Evolução da pressão de impacto na parede (ponto C, ver a fig. 5.1). Os valores obtidos para a pressão no ponto C são comparados com a simulação de [6] e os dados experimentais de [58].

5.3 Discussões e Conclusões

Como pode ser visto nas figuras 5.28 5.29 e 5.30, os resultados obtidos neste trabalho estão coerentes com aqueles encontrados na literatura e, além disso, apresentam uma evolução no tempo mais estável do que os encontrados nas referências [6, 58]. Esta característica é consequência da estabilidade do mSPH.

6 CONCLUSÕES

6.1 Contribuições da dissertação

O surgimento do *Smoothed Particle Hydrodynamics* (SPH) se deu em 1977 [19], no campo de fluxos astrofísicos, com a tarefa de simular modelos estelares politrópicos, *i.e.*, fluxos ilimitados e compressíveis. O SPH é um método numérico, de partículas, lagrangiano e sem malha. Diferentemente dos métodos com malhas, tais como o Método dos Elementos Finitos (na sigla em inglês FEM), o SPH não usa qualquer grade para computar as derivadas espaciais. Em vez disso, são computadas por meio da diferenciação analítica de uma técnica de interpolação. O algoritmo SPH é atrativo por ser simples, eficiente, pois o número de operações é $O(N)$, e altamente paralelizável. Mais ainda, é robusto e consegue capturar a cinemática complexa de fluxos altamente não lineares. Por essas razões, o SPH tem sido usado em diferentes estudos, incluindo fluxos incompressíveis, fluxos altamente compressíveis, explosões, fluxos granulares, captura de ondas de choque e muito mais.

O SPH foi primeiro estendido para simular fluxos hidrodinâmicos com superfície livre [40]. É usado para simular fluxos altamente não lineares, possuindo superfícies livres com múltiplos pontos de conexão (tais como o caso de teste que simula a quebra de uma barragem, ou inglês, *dam-break*) [58], em que a maioria dos outros métodos falham. É fato que a superfície livre simulada tem uma boa comparação com experimentos. Entretanto, a dinâmica fica impregnada de frequências espúrias altamente oscilatórias, com grandes amplitudes, fazendo com que os resultados da dinâmica fiquem comprometidos se não for empregado algum tipo de filtragem.

Apesar do método ser atrativo por apresentar as características positivas citadas, o SPH não pode ser validado para um simples caso de simulação hidrostática [22]. Além disso, a falta de análises conclusivas, seja numérica ou analítica, da

acurácia e consistência do método, somada à instabilidade nas simulações, levou ao emprego de tratamentos semi-empíricos no SPH. O uso de tais tratamentos adicionou incertezas ao método, por exemplo com relação a física simulada [22].

O mSPH surgiu justamente depois de realizada a primeira análise unificada dos erros quantitativos no SPH [22]. O trabalho feito em [22] focou em três objetivos. Primeiro, fazer uma investigação analítica e numérica do método SPH para fluxos com superfície livre, tendo como objetivo principal entender a física capturada e o comportamento numérico através da análise. Segundo, analisar os tratamentos semi-empíricos comumente empregados no SPH, elucidando possíveis benefícios e aplicabilidades. Terceiro, baseado nos dois objetivos anteriores, criar um método SPH convergente que seja capaz de obter uma dinâmica válida, mantendo a simplicidade, robustez e eficiência do SPH. É com este terceiro objetivo que nasce o mSPH.

A técnica de interpolação no SPH é chamada de *Kernel Interpolation* (KI) e sua implementação depende de um algoritmo de busca de vizinhos, *i.e.*, partículas de fluido vizinhas. Esta busca pode ser traduzida para um algoritmo de detecção de colisão. O custo computacional da etapa de detecção de colisão em simulação de partículas é alto o suficiente para dominar o custo computacional total da simulação [25]. Logo, uma implementação eficiente, em termos computacionais, deve atentar para a importância desta etapa.

Em 2010, surgiu um algoritmo de detecção de colisão em GPU que se apresentava 71 vezes mais rápido que o algoritmo considerado estado da arte na detecção de colisão em CPU [25]. Este trabalho serviu como guia e inspiração para grande parte da implementação do mSPH em GPU, em particular a detecção de colisão. Esta etapa é, de longe, a mais complicada de se implementar quando programando em GPU. Apenas para evidenciar esse fato, foi realizada uma implementação da detecção de colisão em CPU de alto desempenho em menos de uma semana, porém demorou-se por volta de sete meses para fazer o algoritmo concorrente em GPU. Mais ainda, a descrição em alto nível de um algoritmo, no domínio da programação

em GPU, é por vezes tão distante da respectiva implementação, que pode tornar este mapeamento um verdadeiro desafio.

O algoritmo apresentado em [25] é bastante genérico e, avaliando as características de uma simulação SPH, percebeu-se que este algoritmo poderia ser especializado. Então apresenta-se nesta dissertação um algoritmo, e sua respectiva implementação em GPU, derivado de [25], que é mais simples de implementar.

Ao tomar conhecimento do trabalho de [22], em particular do método mSPH e suas atrativas características, decidiu-se iniciar uma pesquisa sobre o SPH. A pesquisa permitiu a elaboração desta dissertação, a qual tem como objetivos: i) compartilhar o conhecimento adquirido sobre o SPH e ii) exibir todos os detalhes de uma implementação computacional do mSPH em GPU. O segundo objetivo foi incluído como parte do trabalho devido a dois fatores. Primeiro, o SPH é altamente paralelizável e já foram exibidas, pela comunidade científica, inúmeras simulações mostrando a larga vantagem em termos de desempenho ao rodar o SPH em GPU. Segundo, apesar disso, até onde a se sabe, essa é a primeira vez que se exibe uma implementação do SPH em GPU com tanto detalhes. Talvez por ser um campo muito recente, a implementação de algoritmos em GPU ainda é uma tarefa extremamente árdua. Pior ainda, é muito fácil construir um programa para GPU extremamente ineficiente, mesmo não sendo este o objetivo.

Após muita dedicação, os dois objetivos da dissertação foram alcançados. Nela, está compilado o estudo sobre o SPH, em particular o mSPH, assim como uma descrição detalhada da implementação do mSPH, a qual foi construída ao longo deste trabalho. Tal implementação faz uso do hardware computacional que se encontra cada vez mais presente no meio científico. Em particular, o desempenho desta implementação foi testado em uma única placa de vídeo de baixo custo (a Radeon 6790), comparando-se o resultado com uma outra implementação citada em [6]. Esta última rodando em um *cluster* de 4 processadores. A implementação que fez o uso da placa de vídeo mostrou-se 4x mais rápida em relação a outra (ver seção 4.3).

No capítulo 2 foi descrito o método MA-SPH, o qual é isento de qualquer tratamento semi-empírico. Este método é baseado no SPH apresentado em [40]. O MA-SPH foi considerado pelo trabalho de [22] como o método SPH canônico para fluxos compressíveis e com fronteira livre. A formulação MA-SPH foi o alvo da análise no estudo feito em [22] e serviu como base para a criação do mSPH. A apresentação do MA-SPH, no capítulo 2, inicia-se através da descrição do caso de teste que simula uma quebra de barragem (*dam-break*) como uma motivação. Posteriormente, descreve-se sua discretização espacial, condições iniciais e de fronteira, equações governantes e a técnica de computação das derivadas espaciais KI. Conclui-se o capítulo com discussões sobre os tratamentos semi-empíricos comumente empregados nas simulações SPH.

No capítulo 3, encontra-se a descrição do mSPH de maneira análoga à descrição do MA-SPH, esta última feita no capítulo anterior. Ao final do capítulo 3 é descrita uma regularização do método mSPH.

O capítulo 4 contém a implementação do mSPH em GPU. Inicia-se o capítulo descrevendo primitivas básicas (porém nada triviais de serem implementadas em GPU) utilizadas na implementação. E, finalmente, é exibida a implementação computacional baseada em diferentes níveis de abstrações: i) fluxograma; ii) textual e iii) pseudo-código.

A implementação e o mSPH são validados no capítulo 5. O caso de teste comumente utilizado na validação de uma formulação SPH, quebra de barragem, é descrito. Em seguida, a evolução temporal, através de sequências de imagens, do fluxo e suas variáveis de campo são também exibidas. Também é feito uso de comparações com experimentos reais e numéricos presentes em outros trabalhos.

6.2 Perspectivas no mSPH

O mSPH ainda é muito recente, contudo certamente tem o potencial de contribuir para a evolução do método SPH. Principalmente, aumentar a credibilidade do método perante a comunidade científica, pois não utiliza quaisquer tratamentos

semi-empíricos e foi validada a consistência do KI dentro do domínio.

6.3 Perspectivas na implementação do mSPH em GPU

Como a GPU tem se mostrado um componente chave nas implementações do SPH, acredita-se que a tendência é que esta peça de hardware se torne indispensável nas simulações de fluidos através de métodos de partículas. Os benefícios são óbvios: o uso da GPU acelera os cálculos científicos, acelerando também os resultados das simulações, permitindo, por exemplo, fazer mais experimentos com o objetivo de testar mais hipóteses.

6.4 Trabalhos futuros

A formulação atual do mSPH descarta o termo de viscosidade na equação do momento. Incluir o termo de viscosidade para simular diferentes fluidos, *i.e.*, fluidos viscosos, certamente é um trabalho candidato a estudo futuro. Embora o mSPH tenha sido validado para o caso de teste de quebra de barragem, até onde se sabe, ainda não foram feitos quaisquer testes com fluídos clássicos, tais como *colette* e *poiseuille*. Outros casos de testes podem ser incluídos no mSPH quando a condição de contorno para fronteiras impermeáveis forem estendidas para geometrias mais complexas.

A implementação mSPH não foi otimizada, portanto pode-se fazer um estudo nesta área. O principal fator de otimização que poderá ser usado na implementação apresentada neste estudo é o uso extensivo de memória local, pois praticamente todos os dados usado pelos *kernels* trabalham diretamente com a memória global. Se por um lado o uso da memória global torna mais fácil a implementação, por outro, perde-se em desempenho, pois infelizmente a velocidade de acesso à memória global é muito inferior à velocidade de acesso à memória local. Além disso, a implementação suporta apenas uma GPU, e um trabalho futuro poderia incluir suporte a mais GPUs, ou até mesmo a um conjunto heterogêneo de dispositivos computacionais.

REFERÊNCIAS

[1] BALSARA, D. S. Von Neumann stability analysis of Smoothed Particle Hydrodynamics - suggestions for optimal algorithms. **Journal of Computational Physics**, San Diego, CA, v. 121, n. e, p. 357–372, Oct. 1995.

[2] BARAFF, D. **Dynamic Simulation of Non-Penetrating Rigid Bodies**. 1992. Tese (Doutorado em Ciência da Computação) — Cornell University, Ithaca, Nova Iorque. 1992.

[3] BELYTSCHKO, T. et al. A unified stability analysis of meshless particle methods. **International Journal for Numerical Methods in Engineering**, Hoboken, NJ, v. 48, n. 9, p. 1359–1400, 2000.

[4] BLELLOCH, G. E. **Prefix Sums and Their Applications**. Pittsburgh, PA: Carnegie Mellon University, 1990. (CMU-CS-90-190).

[5] BUCK, I. et al. Brook for GPUs: stream computing on graphics hardware. **ACM Transactions on Graph**, New York, v. 23, n.3, p. 777–786, Aug. 2004.

[6] CHERFILS, J. M. ; PINON, G. ; RIVOALEN, E. JOSEPHINE: a parallel sph code for free-surface flows. **Computer Physics Communications**, Amsterdam, v. 183, n.7, p.1468–1480, Jul. 2012.

[7] CHUNG, T. J. Nonlinear problems/convection dominated flows. In:_____. **Computational fluid dynamics**, Cambridge, UK: Cambridge University Press, 2002, p.347–417p.

[8] CLARK, J. H. The Geometry Engine: a vlsi geometry system for graphics. **ACM SIGGRAPH Computer Graphics**, New York, v. 16, n.3, p.127–133, Jul. 1982.

[9] COHEN, J. D. et al. I-COLLIDE: an interactive and exact collision detection system for large-scale environments. In: SYMPOSIUM ON INTERACTIVE 3D GRAPHICS, 1995, Monterey, CA. **Proceedings ...**, New York: ACM, p.189–196.

[10] COLAGROSSI, A. ; LANDRINI, M. A. Numerical simulation of interfacial flows by SPH. **Journal of Computational Physics**, San Diego, CA, v. 191, n. 2, p.448–475, Nov. 2003.

[11] COLIN, F. ; EGLI, R. ; LIN, F. Y. Computing null divergence velocity field using smoothed particle hydrodynamics. **Journal of Computational Physics**, San Diego, CA, v. 217, n. 2, p.680–692, Sept. 2006.

[12] CUMMINS, S. J. ; RUDMAN, M. An SPH Projection Method. **Journal of Computational Physics**, San Diego, CA, v. 52, n., p.584–607, Jul. 1999.

[13] DALRYMPLE, R. A. ; ROGERS, B. D. Numerical modeling of water waves with the SPH method. **Coastal engineering**, Amsterdam, v. 53, n. 2-3, p.141–147, Feb. 2006.

[14] ERICSON, C. **Real-Time collision detection**. San Francisco: Morgan Kaufmann, 2005. (The Morgan Kaufmann Series in Interactive 3D Technology).

[15] EYLES, J. et al. Pixel-Planes 4: a summary. In: KUIJK, A. A. M. ; STRABER, W. (Eds.). **Advances in computer graphics hardware II Eurographics' 87 Workshop**, Amsterdam: Springer, 1988, p.183–207.

[16] EYLES, J. et al. PixelFlow: the realization. In: ACM SIGGRAPH/EUROGRAPHICS WORKSHOP ON GRAPHICS HARDWARE, 1997, Los Angeles. **Proceedings ...**, New York: 1997, p.57–68.

[17] FUCHS, H. ; POULTON, J. Pixel-Planes: a vlsi-oriented design for a raster graphics engine. **VLSI Design**, Manhasset, NY, v. 2, n. 3, p.20–28, Third Quarter, 1981.

[18] FUCHS, H. et al. Pixel-planes 5: a heterogeneous multiprocessor graphics system using processor enhanced memories. **ACM SIGGRAPH Computer Graphics**, New York, v. 23, n.3, p.79–88, Jul. 1989. Special issue: Proceedings of the 1989 ACM SIGGRAPH Conference.

[19] GINGOLD, R. A. ; MONAGHAN, J. J. Smoothed particle hydrodynamics: theory and application to non-spherical stars. **Monthly Notices of the Royal Astronomical Society**, London, v. 181, p.375–389, Nov. 1977.

[20] HARADA, T. ; KOSHIZUKA, S. ; KAWAGUCHI, Y. Smoothed particle hydrodynamics on GPUs. In: Computer Graphics International, 2007 Petrópolis. **Proceedings ...**, Petrópolis: SBC, 2007.

[21] IGLESIAS, A. S. ; ROJAS, L. P. ; RODRIGUEZ, R. Z. Simulation of anti-roll tanks and sloshing type problems with smoothed particle hydrodynamics. **Ocean Engineering**, Elmsford, NY, v. 31, n. 8-9, p.1169–1192, Jun. 2004.

[22] KIARA, A. **Analysis of the smoothed particle hydrodynamics method for free-surface flows**. 2010. Tese (Doutorado em Ciência da Computação) — Massachusetts Institute of Technology, Cambridge, 2010.

[23] LAMB, H. **Hydrodynamics**. New York: Dover Publications, 1993.

[24] LIEW, K. M. ; HUANG, Y. Q. ; REDDY, J. N. Analysis of general shaped thin plates by the moving least-squares differential quadrature method. **Finite Elements in Analysis and Design**, Amsterdam, v. 40, n. 11, p.1453–1474, Jul. 2004.

[25] LIU, F. et al. Real-time collision culling of a million bodies on graphics processing units. **ACM Transactions on Graphics**, New York, v. 29, n. 6, Article 154:1–154:8, Dec. 2010.

[26] LIU, G. R. ; LIU, M. B. **Smoothed Particle Hydrodynamics**: **a meshfree particle method**. Singapore: World Scientific Publishing, 2003.

[27] LOI, L. **Fast GPU-based Collision Detection**. 2010. Dissertation (Master of Science Thesis in the Programme Computer Science: Algorithms, Language and Logic) — Department of Computer Science and Engineering, de Ciência da Computação e Engenharia, Gotemburg University, Chalmers University of Technology, Gotemburg, 2010.

[28] LONGUET-HIGGINS, M. S. A theory on the origin of microseisms. Philosophical transactions of the Royal Society of London. **Mathematical and Physical Sciences, Series A**, Cambridge, v. 243, n. 857, p.1–35, 1950.

[29] LUCY, L. B. A numerical approach to testing the fission hypothesis. **Astronomical Journal**, Woodbury, v. 82, n. 12, p.1013–1024, 1977.

[30] LUEBKE, D. ; HUMPHREYS, G. How GPU works. **IEEE Computer**, Piscataway, v. 40, n. 2, p.96–100, Feb. 2007.

[31] MACIÀ, F. et al. Benefits of using a Wendland kernel for free-surface flows. In: INTERNATIONAL SPHERIC WORKSHOP ON SPH APPLICATIONS, 6., 2011, Hamburg. **Proceedings ...**, Hamburg: DFG, p.8–10, 2011.

[32] MEI, C. C. **The applied dynamics of ocean surface waves**. Cingapura: World Scientific, 1989.

[33] MELVILLE, W. K. The role of surface-wave breaking in air-sea interaction. **Annual Review Fluid Mechanics**, Palo Alto, v. 28, p.279–321, Jan. 1996.

[34] MERRIL, D. ; GRIMSHAW, A. **Parallel Scan for Stream Architectures**. Charlottesville:: Universy of Virginia, Dez. 2009. (Technical Report CS2009-14).

[35] MERRILL, D. ; GRIMSHAW, A. High performance and scalable radix sorting: a case study of implementing dynamic parallelism for gpu computing. **Parallel Processing Letters**, Singapore, v. 21, n. 2, p.245–272, 2011.

[36] MOLNAR, S. ; EYLES, J. ; POULTON, J. PixelFlow: high-speed rendering using image composition. **ACM SIGGRAPH Computer Graphics**, New York, v. 26, n. 2, p.231–140, Jul. 1992.

[37] MONAGHAN, J. J. ; LATTANZIO, J. C. A refined particle method for astrophysical problems. **Astronomy and Astrophysics**, Les Ulis, v. 149, n. 1, p.135–143, Aug 1985.

[38] MONAGHAN, J. J. On the problem of penetration in particle methods. **Journal of Compuational Physics**, San Diego, CA, v. 82, n. 1, p.1–15, May 1989.

[39] MONAGHAN, J. J. Smoothed Particle Hydrodynamics. **Annual review of astronomy and astrophysics**, New York, v. 30, p.543–574, 1992.

[40] MONAGHAN, J. J. Simulating free surface flows with SPH. **Monthly Notices of the Royal Astronomical Society**, London, v. 110, p.399–406, 1994.

[41] MONAGHAN, J. J. SPH without a tensile instability. **Journal of Computational Physics**, San Diego, CA, v. 159, n. 2, p.290–311, Apr 2000.

[42] MORRIS, J. P. **Analysis of smoothed particle hydrodynamics with applications**, 1996. Thesis (PhD Computer Science) – Monash University, Melbourne, 1996.

[43] MORRIS, J. P. ; FOX, P. J. ; ZHU, Y. Modeling low Reynolds number incompressible flows. **Journal of Computational Physics**, San Diego, CA, v. 136, n. 1, p.214–226, Sept. 1997.

[44] MÜLLER, M. ; CHARYPAR, D. ; GROSS, M. Particle-based fluid simulation for interactive applications. In: Proceedings of the 2003 ACM SIGGRAPH/EUROGRAPHICS SYMPOSIUM ON COMPUTER ANIMATION, 2003. San Diego, CA. **Proceedings ...**, New York: ACM, 2003, p.154–159.

[45] NARAYANASWAMY, M. S. **A hybrid Boussinesq-SPH wave propagation model with application to forced waves in rectangular tanks**, 2008. Thesis (PhD of Philosophy) – Johns Hopkins University, Baltimore, 2008.

[46] NEGRUT, D. et al. **Solving large multi-body dynamics problems on the GPU**. Argonne, IL: Computer Science Division, National Laboratory, Computer Science, 2010. Preprint ANL/MCS-P1777-0710.

[47] OGER, G. et al. Two-dimensional SPH simulations of wedge water entries. **Journal of Computational Physics**, [S.l.], v. 213, p.203–822, 2006.

[48] OGER, G. et al. An improved SPH method: towards higher order convergence. **Journal of Computational Physics**, San Diego, CA, v. 225, n. 2, p.1472–1492, Aug 2007.

[49] OLANO, M. ; LASTRA, A. A shading language on graphics hardware: the pixel-flow shading system. In: ANNUAL CONFERENCE ON COMPUTER GRAPHICS AND INTERACTIVE TECHNIQUES, 25., 1998, Orlando. **Proceedings ...**, New York: ACM, 1998, p.159–168.

[50] OWENS, J. D. et al. A Survey of general-purpose computation on graphics hardware. **Computer Graphics Forum**, Amsterdam, v. 26, n. 1, p.80–113, 2007.

[51] PEERCY, M. S. et al. Interactive multi-pass programmable shading. In: ANNUAL CONFERENCE ON COMPUTER GRAPHICS AND INTERACTIVE TECHNIQUES, 27., 2000, New Orleans. **Proceedings ...**, New York: ACM, p.425–432.

[52] PRICE, D. J. **Magnetic fields in Astrophysics**. 2004. Thesis (Doctor of Philosophy) – Institute of Astronomy an Churchill College, University of Cambridge, Cambridge, 2004.

[53] PURCELL, T. J. et al. Ray tracing on programmable graphics hardware. **ACM Transactions on Graphics**, New York, v. 21, n. 3, p.703–712, Jul. 2002.

[54] STACEY, R. New finite-difference methods for free-surface with a stability analysis. **Bulletin of the Seismological Society of America**, Albany, v. 84, n. 1, p.171–184, Feb. 1994.

[55] STONE, C. P. et al. GPGPU parallel algorithms for structured-grid CFD codes. In: 20th AIAA COMPUTATIONAL FLUID DYNAMICS CONFERENCE, 20., 2011, Honolulu. **Proceedings ...**, Reston, VA: AIAA, 2011, p.2011–3221.

[56] SUTHERLAND, I. E. et al. A Characterization of ten hidden-surface algorithms. **ACM Computing Surveys**, New York, v. 6, n. 1, p.1–55, Mar. 1974.

[57] SWEGLE, J. W. ; HICKS, D. L. ; ATTAWAY, S. W. Smoothed particle hydrodynamics stability analysis. **Journal of Computational Physics**, San Diego, CA, v. 116, n. 1, p.123–134, Jan 1995.

[58] ZHOU, Z. Q. ; KAT, J. O. D. ; BUCHNER, B. A nonlinear 3-D approach to simulate green water dynamics on deck. In: INTERNATIONAL CONFERENCE ON NUMERICAL SHIP HYDRODYNAMICS, 7., 1999, Nantes, FR. **Proceedings ...**, Nantes, FR: Ecole Centrale de Nantes, 1999, p.1–15.

7 APÊNDICE

```
1  Kernel projetaNumColisoes()
       Input: A1, A2, D, X, α, r e k
       Output: C
       /* Se existirem mais threads que partículas */
2      if id < k then
3          {M_a, a} = A1[id];
4          pos_a = X[a];        // Obtém a posição da partícula a
5          cBits_a = D[a];      // Obtém a máscara de bits da partícula a
6          η_a = 0;             // Inicializa o contador de colisões
7          j = 1; // Iterador de partículas candidatas à colisão
8          for (η_a < α) && (id + j < k) do
9              idx_b = id + j; // Índice da partícula candidata b
10             m_b = A2[idx_b];
               /* Checa intersecção de intervalos */
11             if m_b < M_a then
12                 b = A1[idx_b];
13                 cBits_b = D[b];
                   /* As partículas compartilham alguma célula? */
14                 if cBits_a & cBits_b then
15                     pos_b = X[b];
16                     dist = pos_a − pos_b;          // Dist. entre centros
17                     distAoQuad =< dist, dist >; // Produto interno
                       /* Checa colisão */
18                     if distAoQuad < r^2 then
19                         η_a = η_a + 1;
20                     end
21                 end
22             else
                   /* Acabou a busca pois não há mais intersecções */
23                 break;
24             end
               /* Passa para a próxima partícula */
25             j = j + 1;
26         end
           /* Projeta o núm. de colisões encontradas da part. a */
27         C[a] = η_a
28     end
29 end
```

Função projetaNumColisoes - Projeta o número de colisões que cada partícula participa.

```
1  Kernel achaParesColidindo()
       Input: A1, A2, C, D, X, α, r e k
       Output: P1, P2, O
       /* Se existirem mais threads que partículas */
2      if id < k then
3          {M_a, a} = A1[id];
4          η_a = C[a];          // Obtém o número de colisões de a
5          pos_a = X[a];        // Obtém a posição da partícula a
6          cBits_a = D[a];      // Obtém a máscara de bits da partícula a
7          offset_a = O[a];     // Obtém o offset da partícula a
8          numPares = 0;        // Contador de pares encontrados
9          j = 1; // Iterador de partículas candidatas à colisão
10         while numPares < η_a do
11             idx_b = id + j; // Índice da partícula candidata b
12             m_b = A2[idx_b];
               /* Checa intersecção de intervalos */
13             if m_b < M_a then
14                 b = A1[idx_b];
15                 cBits_b = D[b];
                   /* As partículas compartilham alguma célula? */
16                 if cBits_a & cBits_b then
17                     pos_b = X[b];
18                     dist = pos_a − pos_b;          // Dist. entre centros
19                     distAoQuad =< dist, dist >; // Produto interno
                       /* Checa colisão */
20                     if distAoQuad < r^2 then
21                         i = offset_a + numPares;
22                         P1[i] = a;
23                         P2[i] = A1[idx_b];
24                         numPares = numPares + 1;
25                     end
26                 end
27             else
28                 return;
29             end
               /* Passa para a próxima partícula */
30             j = j + 1;
31         end
32     end
33 end
```

Função achaParesColidindo - Acha os pares de colisão, colocando-os em $P1$ e $P2$.

```
1  Kernel calculaFlags()
       Input: S, k
       Output: M
       /* Certamente o índice 0 inicia a primeira sublista */
2      M[0] = 1;
       /* Se existirem mais threads que elementos em S */
3      if id < k then
           /* A primeira thread não participa */
4          if id > 0 then
5              if S[id] ≠ S[id − 1] then
6                  M[id] = 1;
7              else
8                  M[id] = 0;
9              end
10         end
11     end
12 end
```

Função calculaFlags - Calcula o vetor de flags M, marcando a primeira ocorrência de um elemento pertencente a cada sublista.

```
1  Kernel calculaInicioSublistas()
       Input: M, O, k
       Output: I
       /* Se existirem mais threads que elementos em S */
2      if id < k then
3          if M[id] = 1 then
4              I[O[id] − 1] = id;
5          end
6      end
7  end
```

Função calculaInicioSublistas - Calcula o índice inicial de cada sublista, ou seja, o início de cada uma delas.

```
1  Kernel calculaQtdElemSublistas()
       Input: S, M, k
       Output: Q
       /* Somente a última thread não participa */
2      if id < k − 1 then
3          if S[id] ≠ S[id + 1] then
4              posInicial = O[id] − 1;
5              Q[O[id] − 1] = id − posInicial;
6          end
7      else
8          posInicial = O[k − 1];
9          Q[posInicial] = k − posInicial;
10     end
11 end
```

Função calculaQtdElemSublistas - Kernel usado para calcular a quantidade de elementos em cada sublista.

Variáveis de uso geral	
n_r	Número total de partículas reais de fluido.
n_t	Número total corrente de partículas na simulação (reais + fantasmas) .
c	Velocidade artificial do som.
ρ_u	Densidade não perturbada do fluido.
g	Gravidade.
q	Estrutura vetorial contendo as funções de campo (ver fig. 4.12).
Q	Estrutura vetorial contendo as variações temporais das funções de campo (ver fig. 4.13).
wQ	Estrutura vetorial contendo os quatro pesos que multiplicam as derivadas no Runge-Kutta de 4ª ordem (ver sec. 3.2.5).
ΔT	Estrutura vetorial contendo os quatro dt's do Runge-Kutta de 4ª ordem (ver sec. 3.2.5).

Tabela 7.1: Tabela contendo as variáveis globais usadas na descrição da implementação computacional do mSPH. Todas estas variáveis são alocadas no início do algoritmo e no espaço de memória global da GPU.

Variáveis usadas para computar fantasmas	
n_p	Número total de planos. Devido a geometria do domínio, retangular, $n_p = 4$.
n_s	Guarda o número de quantas partículas geraram fantasmas.
n_g	Número máximo de partículas fantasmas que cada partícula real pode gerar em cada passo de tempo. Devido a geometria do domínio, retangular, $n_g = 3$.
α	Constante considerada na condição de fronteira (ver sec. 3.2.3).
P	Estrutura vetorial contendo a representação dos planos, sendo que $P[i]$ armazena os coeficientes da equação vetorial do plano i.
N_f	Estrutura vetorial contendo na posição i a quantidade de partículas fantasmas geradas em cada passo de tempo pela partícula real i.
O_f	Estrutura vetorial contendo o resultado da soma prefixal (exclusiva) de N_f.
M_f	Matriz de tamanho $2 \times n_r$ para guardar índices de planos. Se a partícula i está próxima o suficiente do plano j e $j+1$ para criar duas partículas fantasmas, então $M_f[i][0] = j$ e $M_f[i][1] = j + 1$.
C_f	Estrutura vetorial contendo o id's das partículas que geram alguma partícula fantasma.
D_f	Estrutura vetorial contendo somente 0's e 1's. Se a partícula 1 gera alguma partícula fantasma, então $D_f[i] = 1$, caso contrário, $D_f[i] = 0$.

Tabela 7.2: Tabela contendo as variáveis globais usadas na descrição da implementação computacional do mSPH. Todas estas variáveis são alocadas no início do algoritmo e no espaço de memória global da GPU.

```
  /* Função responsável por criar partículas fantasmas a cada
     iteração.  */
1 Function preparaFantasmas()
    // Abaixo, as chamadas de 5 kernels
2   contaFantasmas(); // Preenche N_f, M_f, C_f, D_f.
3   radixSort(C_f);    // Agrupa partículas que geraram fantasmas.
    /* Conta a quantidade de 1's em D_f, ou seja, a quantidade de
       partículas reais que geraram algum fantasma.  */
4   n_s = reduction(D_f);
    // Cálculo dos índices para gravar em q.
5   O_f = scanExclusivo(N_f);
    // Preenche q com inf.  dos fantasmas usando n_s threads.
6   criaFantasmas();
    /* Calcula o número total de partículas, reais + fantasmas,
       n_t, para determinar a quantidade de threads usadas no
       algoritmo de detecção de colisão.  */
7   numFantasmas = N_f[n_r - 1] + O_f[n_r - 1];
8   n_t = n_r + numFantasmas;
9 end
```

Função preparaFantasmas - Função composta por cinco *kernels*, é responsável por preparar as partículas fantasmas a cada iteração da integração temporal.

```
1  Kernel contaFantasmas()
       Input: n_r, n_g, q, P, n_p, α, h
       Output: N_f, M_f, C, D
       /* Se existirem mais threads que partículas reais.  */
2      if id < n_r then
3          private i = 0;                  // Iterador de planos.
4          private numFantasmas = 0;   // Guarda o número de
               fantasmas.
           /* Percorre todos os planos */
5          while (i < n_p) && (numFantasmas < n_g) do
               /* Verifica distância da partícula à fronteira.  */
6              if distancia( P[i], q[id].X ) ≤ alfa * h then
7                  M_f[numFantasmas][id] = i;
8                  numFantasmas++;
9              end
10             i++;
11         end
12         if numFantasmas > 0 then
13             C_f[id] = id;
14             D_f[id] = 1;
               /* Verifica se precisa criar o fantasma de quina da
                  fronteira.  */
15             if numFantasmas == 2 then
16                 numFantasmas++;
17             end
18         else
               /* Marca com o maior inteiro possível sem sinal(32bits,
                  x86).  Porém, serve qualquer valor maior que o maior
                  id.    */
19             C_f[id] = 0xffffff;
20             D_f[id] = 0;
21         end
           /* Copia resultado final.  */
22         N_f[id] = numFantasmas;
23     end
24 end
```

Função contaFantasmas - Kernel responsável por contar o número de fantasmas ativos. Cada partícula i tem o seu número de fantasmas guardado na posição de índice i em N_f pela *thread* de $id = i$.

```
1  Kernel criaFantasmas()
       Input: n_r, n_g, n_s, N_f, O_f, g, c, P, n_p, α, h, ρ_u
       Output: q
       /* Se existirem mais threads que partículas geradoras de
          fantasmas.  */
2      if id < n_s then
3          private id_r = C_f[id];  // Guarda o id da partícula real.
4          private fantasma = 0;  // Iterador de fantasmas.
5          private delta = |g|/(c * c);  // Para o cálculo de densidade.
6          private normal_q = {0};    // Normal da quina.
7          private x_q = {0};         // Pos.  da fantasma na quina.
8          private offset = n_r + O_f[id_r];
           /* Percorre a coluna M_f[id_r] */
9          while fantasma < N_f[id_r] do
10             private idxPlano = M_f[id_r][fantasma];
11             private normal = P[idxPlano].normal;
12             private id_f = offset + fantasma;
13             private dist = | < q[id_r].X, normal > |;
               /* Cálculo da posição e densidade.  */
14             q[id_f].X = q[id_r].X + 2 * dist * normal;
15             q[id_f].ρ = q[id_r].ρ + ρ_u * delta * (q[id_r].X_y − q[id_f].X_y);
               /* Divide a vel.  em duas componentes ortogonais.  */
16             U_n = < normal, q[id_r].U > *normal;
17             U_t = q[id_r].U − U_n;
               /* Cálculo da velocidade.  */
18             q[id_f].U = U_t − U_n;
               /* Guarda inf.  para possível fantasma da quina.  */
19             normal_q += normal;
20             x_q += q[id_f].X − q[id_r].X;
21             fantasma++;
22         end
           /* Verifica a necessidade de criar fantasma da quina.  */
23         if N_f[id_r] > 1 then
               /* Guarda informações da fantasma da quina.  */
24             ...
25         end
26     end
27 end
```

Função criaFantasmas - Kernel responsável por preencher o vetor q com as informações das partículas fantasmas.