TÉCNICAS DE ORDENAÇÃO PARA SIMULAÇÃO FÍSICA BASEADA EM PARTÍCULAS

Rubens Carlos Silva Oliveira

Dissertação de Mestrado apresentada ao Programa de Pós-graduação em Engenharia de Sistemas e Computação, COPPE, da Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários à obtenção do título de Mestre em Engenharia de Sistemas e Computação.

Orientador: Claudio Esperança

Rio de Janeiro
Setembro de 2012

TÉCNICAS DE ORDENAÇÃO PARA SIMULAÇÃO FÍSICA BASEADA EM PARTÍCULAS

Rubens Carlos Silva Oliveira

DISSERTAÇÃO SUBMETIDA AO CORPO DOCENTE DO INSTITUTO ALBERTO LUIZ COIMBRA DE PÓS-GRADUAÇÃO E PESQUISA DE ENGENHARIA (COPPE) DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS E COMPUTAÇÃO.

Examinada por:

Prof. Claudio Esperança, Ph.D.

Prof. Ricardo Guerra Marroquim, D.Sc.

Prof. Waldemar Celes Filho, Ph.D.

RIO DE JANEIRO, RJ – BRASIL
SETEMBRO DE 2012

Oliveira, Rubens Carlos Silva

Técnicas de Ordenação para Simulação Física Baseada em Partículas/Rubens Carlos Silva Oliveira. – Rio de Janeiro: UFRJ/COPPE, 2012.

XIII, 87 p.: il.; 29, 7cm.

Orientador: Claudio Esperança

Dissertação (mestrado) – UFRJ/COPPE/Programa de Engenharia de Sistemas e Computação, 2012.

Referências Bibliográficas: p. 49 – 54.

1. Técnicas de Ordenação. 2. Simulação Física. 3. Partículas. I. Esperança, Claudio. II. Universidade Federal do Rio de Janeiro, COPPE, Programa de Engenharia de Sistemas e Computação. III. Título.

À vida

Agradecimentos

Agradeço aos meus professores, que são todos referências em inteligência, razoabilidade e benevolência. Em particular, ao professor Claudio Esperança, que, brilhante como é, soube ser um excelente orientador percebendo minhas limitações e qualidades, otimizando, assim, meu rendimento no desenvolvimento desse trabalho. Sem o professor Claudio e sua habilidade diplomática, eu não concluiria em tempo esta dissertação.

Aos meus colegas do mestrado, que tornaram os dias na UFRJ mais divertidos e, justiça seja feita, a todos os demais colegas do LCG, que conseguem aliar muito bem a inteligência ao bom humor.

À Marinha do Brasil e, em particular, aos servidores militares e civis da DIRETORIA DE COMUNICAÇÕES E TECNOLOGIA DA INFORMAÇÃO DA MARINHA (DCTIM) e do CENTRO DE APOIO A SISTEMAS OPERATIVOS (CASOP), que possibilitaram a importante e prazerosa oportunidade de aprimoramento através dos maiores cientistas deste país.

Ao meu pai, que me ensinou a conviver bem com os livros de matemática e física e que também me deu um presente que me incentivou a pensar: um microcomputador CP-200S.

À minha mãe, que sempre esteve ao meu lado me apoiando desde sempre, inclusive abrindo mão dos seus próprios objetivos, priorizando a minha formação e as das minhas irmãs.

A tantas outras pessoas que, direta ou indiretamente, ajudaram a construir meu presente.

E, é claro, ao Criador, sem o qual nada seria possível.

Resumo da Dissertação apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Mestre em Ciências (M.Sc.)

TÉCNICAS DE ORDENAÇÃO PARA SIMULAÇÃO FÍSICA BASEADA EM PARTÍCULAS

Rubens Carlos Silva Oliveira

Setembro/2012

Orientador: Claudio Esperança

Programa: Engenharia de Sistemas e Computação

Vários trabalhos recentes sobre simulação física de objetos rígidos, deformáveis e fluidos têm como idéia central a representação dos corpos discretizados em partículas. Esquemas deste tipo visam aproveitar o fato de que cada partícula é submetida a um processamento muito semelhante para se valer de arquiteturas computacionais paralelas, quer usando múltiplos núcleos, quer usando GPUs. Em grande parte destas propostas, uma etapa crucial e de grande complexidade computacional é aquela relacionada a um procedimento de ordenação das partículas com referência a um espaço de endereços espaciais, procedimento este que precisa ser repetido a cada passo da simulação. O presente trabalho foca precisamente esta etapa, apresentando e avaliando diversos esquemas que tentam aproveitar as características de coerência espacial e temporal das simulações com partículas para obter maior eficiência no processo de ordenação.

Abstract of Dissertation presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Master of Science (M.Sc.)

SORTING TECHNIQUES FOR PHYSICALLY BASED PARTICLE SIMULATION

Rubens Carlos Silva Oliveira

September/2012

Advisor: Claudio Esperança

Department: Systems Engineering and Computer Science

Several recent works on physically based simulation of fluids, deformable and rigid objects use bodies discretized as particles. Such schemes take advantage of the fact that all particles are submitted to a very similar computational treatment, and thus are amenable to implementation in parallel architectures, either multi-core or GPU-based. In many approaches, a crucial and very costly stage is related to sorting the particles with respect to a spatial grid, a procedure that must be repeated for every time step of the simulation. The present work focuses on this particular stage, presenting and evaluating several schemes for taking advantage of spatial and temporal coherence of typical particle simulations in order to obtain performance improvements in the sorting process.

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Na Computação Gráfica, a animação baseada em física (*Physically Based Animation*) é um processo que permite obter a cinemática de objetos com plausibilidade física. Para esse tipo de animação ser considerada satisfatória, ela deverá prever a interação entre os corpos em conformidade com as leis da Física, ou seja, esse tipo de animação é uma imitação de um processo da vida real, que é uma definição de simulação [6]. Essas simulações podem se apresentar de várias formas: simulação de sistemas de partículas, simulação de objetos rígidos, simulação de objetos deformáveis, simulação de fluidos (líquidos, fogo, fumaça etc.) e o acoplamento destas. Esse tipo de animação é empregado em jogos interativos, simulações cirúrgicas, robótica, indústria do cinema, indústria automotiva, entre outros.

Dentre as várias possibilidades de simulações físicas (térmicas, eletromagnéticas,...), as simulações mecânicas costumam ser as de maior interesse em animações, já que possuem reflexo direto no movimento dos corpos. Nesse escopo, o contato é o tipo de interação que "mais" se destaca na mudança do estado de movimento dos corpos. O contato é conseqüência direta da propriedade de dois ou mais corpos não poderem ocupar o mesmo lugar ao mesmo tempo, e essa propriedade da matéria[1] dá origem ao processo computacional conhecido como *detecção de colisão* (DC). A DC também é importante onde interações de *domínio compacto* ocorrem, como por exemplo, em simulações baseadas em SPH [7, 8]. Nelas, a obtenção de um estado físico associado a uma determinada posição no espaço (densidade do fluido no SPH, por exemplo) é realizada através de uma função de base radial que contabiliza a contribuição de amostras vizinhas até uma determinada distância. A fase de DC, por não ser um problema trivial, é onde tipicamente se emprega a maior parcela do esforço computacional. Por isso, a detecção de colisão tem sido uma área merecedora de intensa atividade por parte da comunidade de pesquisadores no intuito de melhorar a precisão e diminuir o tempo gasto nesse processo. Essa fase se torna ainda mais crítica quando se objetiva animações a taxas interativas, o que acaba

[1] Pelo menos em termos de Física Clássica.

motivando também a adoção de hardware poderoso [9].

Nos últimos anos, com a evolução das unidades de processamento gráfico (GPUs), novas técnicas de DC foram apresentadas (por exemplo, [10, 11]), explorando o *espaço-imagem*, contrastando com as técnicas tradicionais [12, 13] que usam o espaço do objeto.

Entre as que trabalham no espaço de objetos, técnicas baseadas em partículas têm se tornado bastante populares nos últimos anos [7, 14]. A base destas técnicas é a representação de objetos por conjuntos de partículas e a detecção de colisão simplesmente se resume a detectar interferência entre pequenas esferas. Estas técnicas se apóiam no uso de uma **grade 3D**[2].

A grade é uma parte fundamental do método, já que permite uma detecção de colisão rápida, baseada no princípio de que uma partícula não pode colidir com outras além daquelas residentes na mesma célula (*voxel*) ou em células adjacentes. O método tradicionalmente se vale de uma etapa de ordenação para obter objetos residentes numa dada célula. O presente trabalho foca nas simulações baseadas em partículas, apresentando sugestões que melhoram o desempenho aproveitando a coerência temporal-espacial na etapa de ordenação. O capítulo 2 apresenta conceitos e principais trabalhos relacionados com detecção de colisão, simulação baseada em partículas e conceitos de programação em GPUs com métodos de ordenação de pares mais empregados nesse tipo de processamento. No capítulo 3, são apresentados aspectos e técnicas que possibilitam a melhora na velocidade da detecção de colisão quando fazendo uso da coerência temporal e espacial nas animações baseadas em partículas. No capítulo 4, os experimentos demonstrativos de desempenho são expostos juntamente com os contextos em que foram conduzidos. No capítulo 5, são apresentadas as conclusões desta dissertação, ficando reservado para o capítulo 6 algumas possibilidades de trabalhos complementares.

[2]Estrutura de dados usada para subdividir o espaço em células regulares.

Capítulo 2

Trabalhos Relacionados

Neste capítulo, são apresentados alguns conceitos importantes para o trabalho desenvolvido nesta dissertação: conceitos de computação paralela e a Unidade de Processamento Gráfico, métodos de detecção de colisão e sua aplicação em métodos de animação baseada em física.

No escopo de detecção de colisão, as técnicas são classificadas em duas categorias: as técnicas que usam o espaço do objeto e as técnicas que usam o espaço da imagem. Essas técnicas possuem ainda implementações sequenciais, que são executadas em CPU, e implementações para processamento paralelo, que podem ser executadas na CPU [8] (com múltiplos núcleos de processamento) ou na GPU. Já na área de animação baseada em física, são revisados trabalhos que usam partículas para representar os objetos.

2.1 Detecção de colisão

A detecção de colisão (DC) é uma tarefa fundamental e um importante ingrediente em muitos aplicativos em Computação Gráfica[15]. Os aplicativos que possuem a DC como papel vital incluem CAD/CAM, modelagem fisicamente baseada, jogos eletrônicos, realidade virtual, planos de montagem, deslocamentos de robôs, simuladores (vôo, automobilismo e outros). Seu objetivo principal pode ser resumido nas respostas a estas questões: se houve colisão, em que instante e onde o contato geométrico entre dois objetos ocorreu [13]. Finalmente, após a caracterização da colisão, é procedida a etapa de resposta à colisão, que vem a ser a tomada de ação decorrente (separação de objetos, aviso ao projetista sobre tolerâncias, resposta mecânica etc.). Em muitas dessas áreas, a DC é considerada o principal gargalo de processamento. Como são diversas as necessidades da DC, várias técnicas foram propostas e cada contexto de emprego vai sugerir a adoção de alguma ou abandono de outra. Critérios como velocidade, precisão e representação dos objetos envolvidos (veja Fig. 2.1) serão determinantes nessa escolha. Em animação baseada em física, tem-se optado por técnicas que possam fornecer dados que facilitem

o processamento da resposta à colisão, o que compreende o cômputo de forças de repulsão[16]. Outro atributo que tem pesado bastante na escolha da técnica de DC tem sido a sua adaptabilidade para a execução em paralelo.

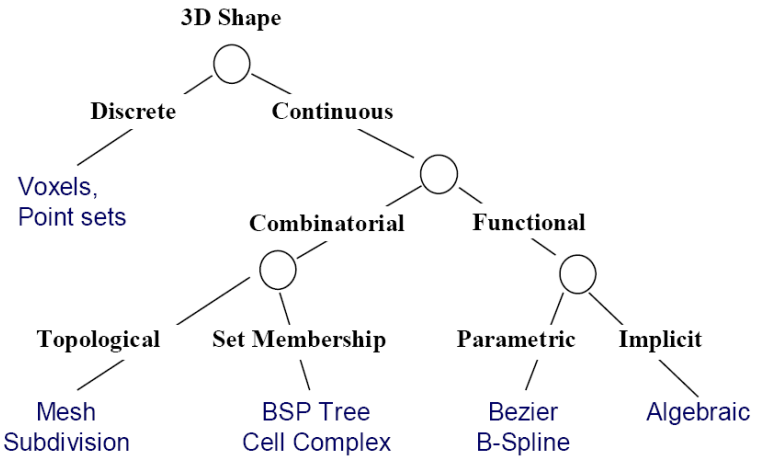

Figura 2.1: Representação dos objetos tridimensionais segundo a taxonomia proposta por Naylor (figura retirada de [1])

2.1.1 Tipos de perguntas a se fazer aos métodos de DC

A pergunta mais simples que pode ser feita na DC é se dois corpos se tocam. Dependendo da aplicação, outras informações poderão ser requeridas: objetos envolvidos, a maior distância e direção na penetração detectada etc. Tais informações podem ser usadas, por exemplo, na reação física de afastamento por penalidade (veja Fig. 2.2).

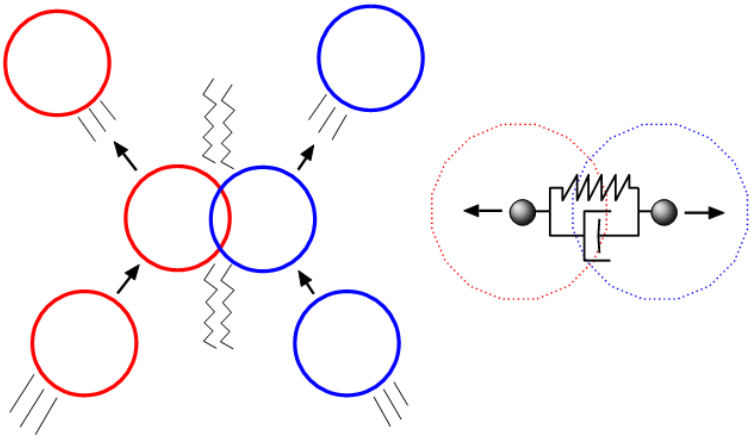

Figura 2.2: Exemplo de afastamento por penalidade via acoplamento elástico e dissipativo

2.1.2 Etapas da DC

Visando diminuir os testes de interferência, a detecção de colisão frequentemente é dividida em duas etapas. A primeira é uma detecção de colisão grosseira (*broad phase*,

em inglês), cujo objetivo não é a exatidão dos resultados, mas apenas o descarte de testes envolvendo objetos que garantidamente não colidam (veja Fig. 2.3). A segunda é a detecção de colisão mais precisa (*narrow phase*), que permite detectar "se", "quando" e "onde" ela ocorreu [17]. Esta etapa frequentemente é a mais custosa, já que depende muito da complexidade geométrica dos objetos em colisão.

A "varredura e poda" é uma técnica de *broad phase* que tem na sua base a projeção dos extremos do subespaço ocupado pelo objeto (representados através de AABB, por exemplo) sobre os eixos de coordenada; possibilitando, dessa forma, o rápido descarte de testes entre objetos que não possuam intersecção nos referidos eixos. Existem também para a *broad phase* técnicas que usam estruturas hierárquicas. As técnicas fundamentadas nessas estruturas de dados costumam perder eficiência quando usadas em cenários muito dinâmicos [18]. Para estes, uma solução bastante empregada é a subdivisão espacial em grade regular. Tal grade, além de ser de simples implementação, é muito apropriada ao processamento nas modernas placas de vídeo. Seu ponto fraco é não lidar bem com distribuições pouco uniformes de objetos no espaço [19].

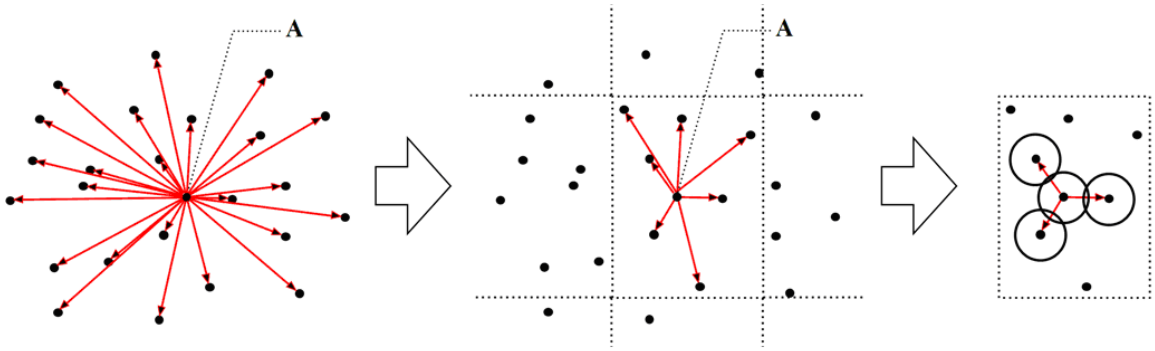

Figura 2.3: Subdivisão espacial para diminuir a quantidade de testes

Para o mesmo fim, existem também técnicas baseadas em código de *hash*. Elas promovem uma associação direta dos volumes limitantes a um determinado *hash* (ou conjunto de), a partir do qual apenas aqueles objetos que compartilham do mesmo código de *hash* serão confrontados no estágio seguinte do estudo de colisão.

A *narrow phase* é a fase de teste responsável por fornecer informações detalhadas da possível colisão (colisão não descartada na *broad phase*) [19]. Essas informações poderão ser: tempo na qual a colisão se deu (caso tenha ocorrido), distância de separação (ou penetração), pontos mais próximos etc. Estas poderão, mais tarde, ser usadas para cálculo de reação, tempo de impacto, entre outros. Em [19], também são apresentados os agrupamentos e técnicas gerais da *narrow phase*. Na presente dissertação, pelo fato de a simulação envolver partículas confinadas a um espaço subdivido em grade, a técnica de *broad phase* empregada foi baseada em tabela de *hash* e a de *narrow phase* foi baseada em volumes limitantes [19], especificamente, esferas.

A adoção de esferas como volumes limitantes a serem analisados traz decorrências

principalmente de velocidade e simplicidade ao algoritmo a realizar tal tarefa. Sua aplicação também pode ser ampliada para tratar corpos extensos [2].

2.1.3 Detecção discreta e detecção contínua

As técnicas de DC podem ser divididas em dois tipos: discreta e contínua [20]. A discreta se baseia na amostragem dos objetos em suas trajetórias, decorridos intervalos de tempos discretos (*time step*). Apenas após a obtenção da interferência entre eles, serão processadas as ações decorrentes da colisão, ou seja, é um teste, *a posteriori*, da interpenetração ([21]). Além dos problemas relacionados ao realismo físico decorrentes da interpenetração, esse tipo de DC pode perder colisões quando aplicada a simulações envolvendo corpos muito finos ou muito rápidos. Técnicas de *backtracking* (volta na trajetória) tentam corrigir (ou diminuir) os problemas de interpenetração da DC discreta, que se dará através de um processo iterativo em que se busca um instante no qual se possa assumir como o tempo do primeiro contato. O meio de se evitar o *backtracking* é admitir a interpenetração entre objetos, o que pode, inclusive, gerar instabilidade em simulações dinâmicas. A DC contínua, resumidamente, é o tipo de técnica que, usando os estados cinemáticos dos objetos da simulação, estimará as características do primeiro contato, ou seja, é um teste *a priori* da interpenetração [21].

2.2 Detecção de colisão em sistemas de partículas

A palavra "partícula" pode ser definida como "corpo material de dimensões muito pequenas" [22]. O termo "dimensões muito pequenas" diz respeito ao tamanho do corpo perante o domínio onde ocorre o fenômeno. Na DC entre partículas, é comum assumir que a interação com outro corpo se dá apenas num espaço limitado por um "raio de ação" (distância limite); ou seja, dois corpos interagem apenas se as esferas que os representam se tocam. Fora as interações semelhantes às gravitacionais, todas as demais (geralmente, as de contato) apenas exercem seus efeitos numa vizinhança bastante limitada, o que torna a adoção da esfera razoável para boa parte das simulações mecânicas. Dessa forma, o teste de DC entre partículas pode ser descrito como na equação da figura 2.4, o que torna o teste de colisão muito simples de ser implementado, já que a esfera é invariante perante as rotações. Outra qualidade decorrente dessa abordagem é sua eficiência.

São várias as aplicações dessa abordagem em física e em engenharia, devido ao seu poder de generalização, possibilitando, inclusive, a unificação da DC entre corpos pontuais e extensos, tanto rígidos como não rígidos. As simulações baseadas em partículas que envolvem sólidos e fluidos são bons exemplos desse poder. Uma forma de generalização do teste de colisão de estruturas complexas é a associação de objetos descritos por malhas poligonais a uma descrição por partículas (Figuras 2.5 e 2.6).

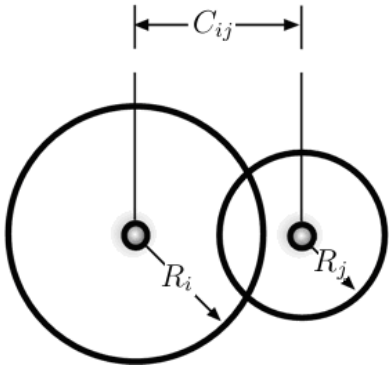

Figura 2.4: Condição para existência de colisão: $R_i + R_j > C_{ij}$

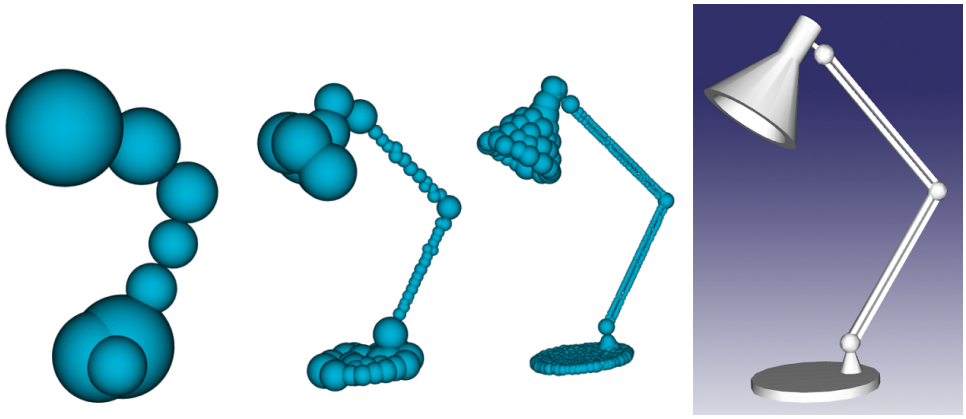

Figura 2.5: Composição de esferas em lugar da malha tradicional (figuras e malha obtidas em [2])

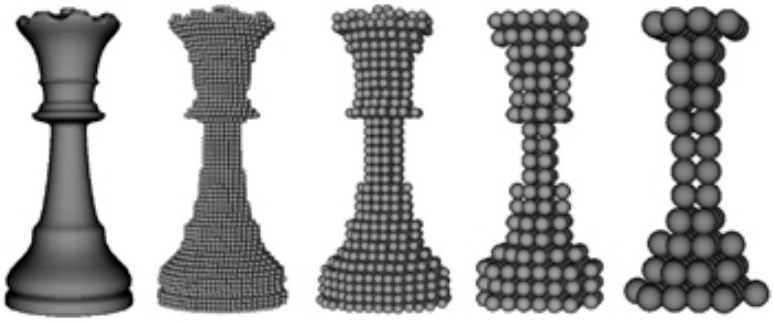

Figura 2.6: Exemplo de várias resoluções na representação de objetos através de esferas (figura obtida em [3])

2.3 DC em sistemas de partículas usando grade regular

Em muitas ocasiões, as simulações envolvendo partículas, para serem consideradas satisfatórias, deverão contar com uma grande quantidade delas (na ordem de milhões). A solução mais simples para DC, que compara todas as partículas entre si, resultaria em esforço quadrático com relação ao número de partículas $O(n^2)$, o que inviabilizaria boa parte das animações interativas. Essa constatação impõe a necessidade de se promover

um eficiente sistema de detecção de colisão para esse tipo de simulação. Em [16], várias técnicas de subdivisão espacial são mencionadas e comparadas segundo contextos de uso. Dentre todas, a grade regular, favorecida pela simplicidade de acesso aos itens que nela se encontram e pela rápida construção da estrutura de dados, tem sido, majoritariamente, a escolhida para uso na *broad phase* desse tipo de simulação.

A grade regular consiste basicamente em subdividir um domínio do espaço através de espaçamentos regulares nos eixos cartesianos. Este processo dá origem a caixas de mesmas dimensões, denominadas células (exemplo tridimensional na figura 2.7).

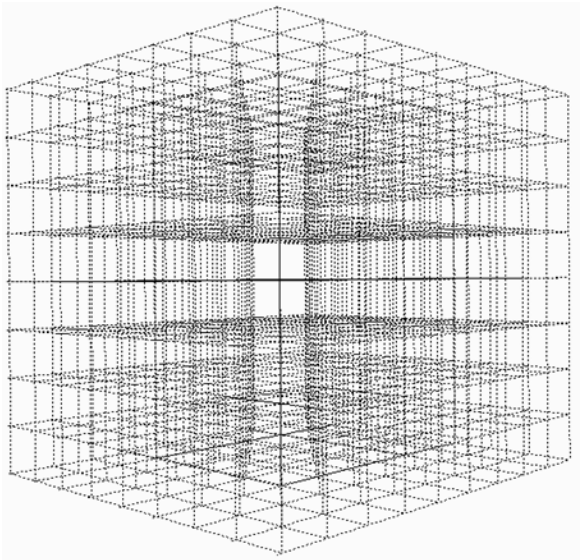

Figura 2.7: Representação da estrutura de grades uniformes

Essa partição do espaço permite a pesquisa de colisão (*narrow phase*) com um subconjunto bem reduzido de partículas, isto é, apenas com partículas situadas nas células vizinhas e na própria célula da partícula de interesse.

A localização da partícula obedecerá ao esquema "loose grid" [23], ou seja, a partícula é dita pertencente a uma determinada célula, apenas se a última contiver o centro da esfera associada (Fig. 2.8).

A implementação da estrutura se divide praticamente em dois tipos: um baseado em compartimentos (*buckets*), onde cada célula comportaria um número máximo de partículas[1]; e outro baseado em ordenação (*sort*), onde um *array* contendo as partículas com respectivos endereços de célula (código de *hash* da célula na qual se encontra), sofreria um processo de ordenação usando o código de hash como chave, que resultaria em reunião das partículas de mesmo hash, ou seja, associadas a uma mesma célula. Essa abordagem tem a vantagem de não impor limite de população às células.

[1] A limitação pode ser superada pela contagem de itens que ocupam cada célula e posterior alocação do espaço requerido.

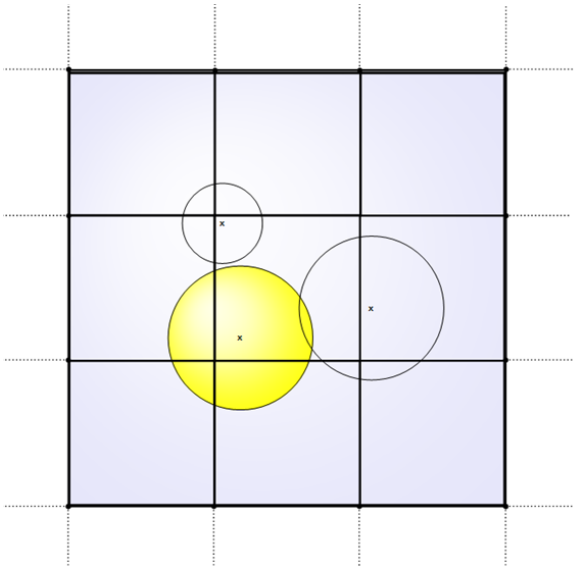

Figura 2.8: Exemplo do esquema “loose grid” e de colisão

2.4 Definição da arquitetura da grade e cálculo do hash da célula

A implementação da estrutura de grade pode ser dividida nos seguintes passos:

1. Delimitação do domínio do espaço: para que se use a estrutura de grade, o primeiro passo é definir o domínio do espaço a ser utilizado na simulação. Esse volume tem a forma de uma caixa.

2. Subdivisão do domínio do espaço: definida a caixa, procede-se com o dimensionamento das células. Mas, visando ao confinamento dos testes às células adjacentes e à célula da partícula de interesse, cada célula terá de ter dimensões que comportem completamente a maior partícula envolvida na simulação [2]

3. Formulação do *hash* da célula: para um eficiente funcionamento da estrutura de grade, faz-se necessária que seja garantida a unicidade do *hash* para cada célula de forma a evitar a reunião desnecessária de partículas que não pertençam à célula de interesse, porque dentro de cada célula todas as partículas são testadas entre si.

Para isso, associado ao rápido cálculo e ao emprego direto do *hash* como índice para um *array* de células, a formulação do tipo “campo de bits” se torna oportuna. Mas antes, seria necessária a obtenção das coordenadas da partícula em termos de grade (i, j e k). Por exemplo, definidas as quantidades de subdivisões dos eixos x, y e z como 32, 64 e 128, respectivamente, o *hash* das células poderia ser calculado por:

[2] Num mesmo sistema a ser simulado, diferentes dimensões de partículas podem coexistir, mas não é rara a opção por todas elas possuírem mesma dimensão para a esfera de teste associada. Isso traz simplificações na implementação dos algoritmos (e mais velocidade).

$$Hash(i, j, k) = i + j * 32 + k * 2048$$

ou

$$Hash(i, j, k) = (i) + (j << 5) + (k << 11)$$

(sintaxe da linguagem "C", na qual o operador $<<$ indica a operação de *shift* de bits à esquerda)

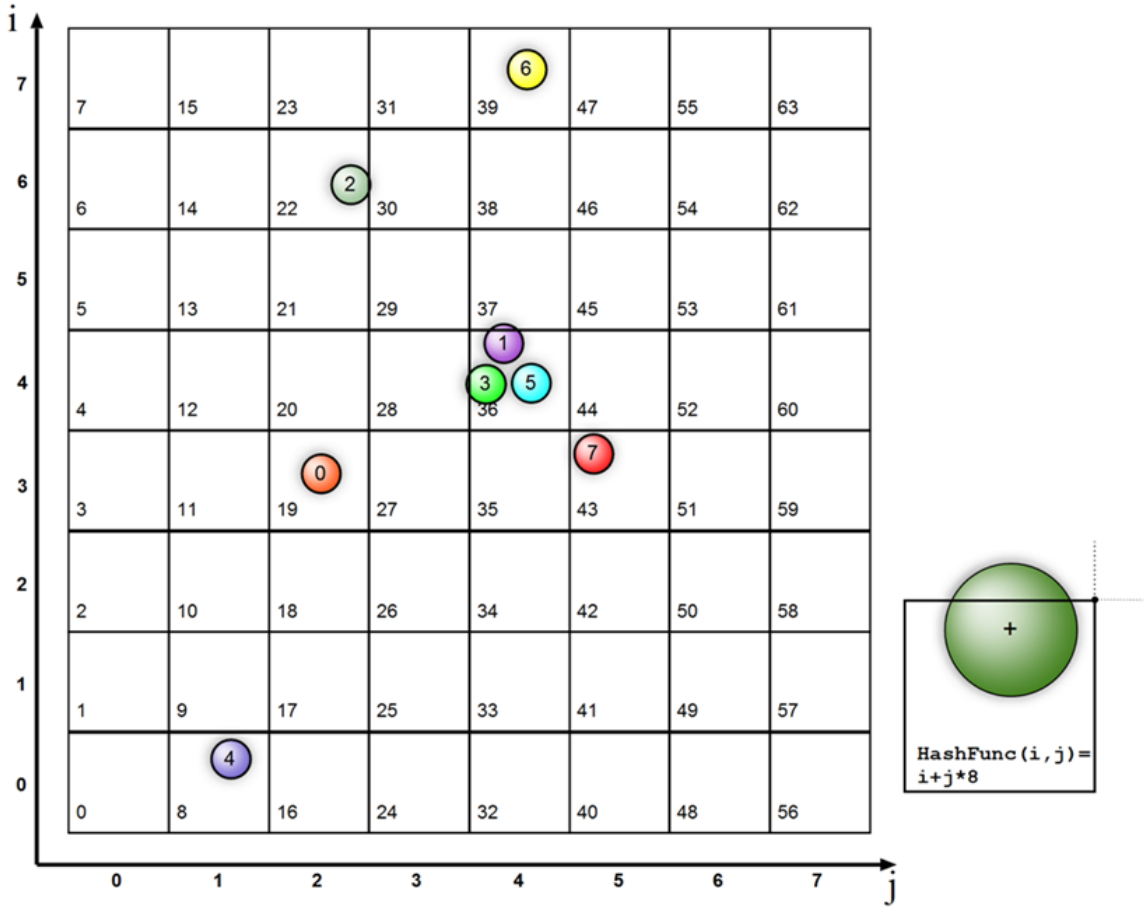

Figura 2.9: Grade e associação do código de *hash* à célula

A obtenção das coordenadas de grade (i, j, k) para uma determinada partícula "n" de centro (X_n,Y_n,Z_n) obedece à seguinte formulação para o espaço:

$$(i_n, j_n, k_n) = \left(\lfloor \frac{X_n - X_)}{L_X} \rfloor, \lfloor \frac{Y_n - Y_0}{L_Y} \rfloor, \lfloor \frac{Z_n - Z_0}{L_Z} \rfloor \right),$$

onde X_n,Y_n,Z_n são coordenadas da partícula n; X_0, Y_0 e Z_0, as coordenadas da referência no espaço e L_X, L_Y e L_Z, as dimensões das células nos eixos x,y e z. Assumindo, por exemplo, que uma simulação de um sistema de partículas no plano aconteça no retângulo limitado pelos vértices $(1.0, 2.0)$ e $(38.0, 78.0)$, e admitindo que os eixos x e y desse

Partícula	Hash Associado
1	36
2	22
3	36
4	8
5	36
6	39
7	43

Tabela 2.1: Partícula e respectivo hash

retângulo sejam divididos em 8 partes (cada eixo com 8 marcas equidistantes), as coordenadas de grade i e j para uma partícula de centro $(13.0, 17.0)$ seriam $(2, 1)$, uma vez que as dimensões da célula L_X e L_Y seriam, respectivamente, $(38.0-1.0)/8.0$ e $(78.0-2.0)/8.0$, que resultaria em 4.625 e 9.5, e as coordenadas da partícula, em termos de grade, seriam dadas por $(i, j) = (\lfloor(13.0 - 1.0)/4.625\rfloor, \lfloor(17.0 - 2.0)/9.5\rfloor)$. Se a função de *hash* fosse dada por $Hash(i, j) = i + 8 * j$, o código de *hash* dessa partícula seria 10. A partir da figura 2.9, é possível a obtenção da tabela 2.1, que relaciona as partículas ao respectivo código de *hash*, também chamada de tabela de pares .

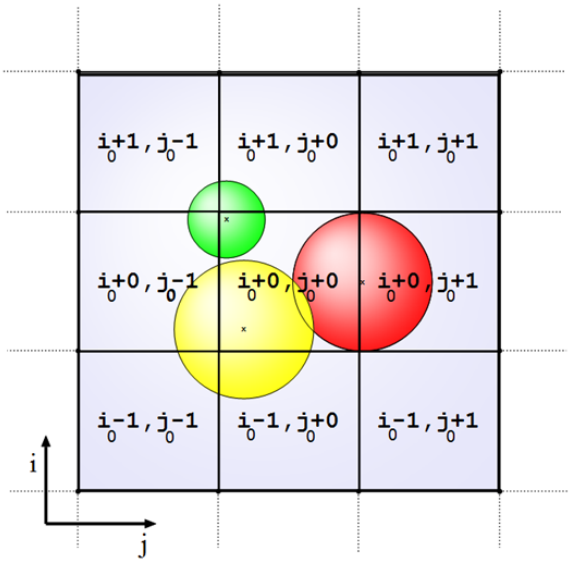

Figura 2.10: Exemplo das coordenadas dos voxels em relação a um voxel central de coordenadas i_0 e j_0

Como observação pertinente, ainda que o $Hash(i, j)$ fosse dado por $i + 32 * j$, por exemplo, também estaria garantida a relação biunívoca entre *hash* e célula (em se respeitando os limites do espaço da simulação); mas, como o índice i está confinado ao intervalo de 0 a 7 (pelas 8 subdivisões adotadas nesse eixo), diversos índices (i de 8 a 31) jamais seriam utilizados. Isto resultaria em desperdício no dimensionamento do *array* que fizesse uso direto dessa função de *hash* como indexador de acesso a alguma informação de célula.

2.5 Ordenação de pares (partículas-hash)

Na figura 2.10, atendidos os requisitos dimensionais entre partículas e células para pesquisa até células adjacentes, observa-se que a busca de colisão para uma dada partícula de coordenadas de grade (i_0, j_0) deverá ser realizada com todas as outras partículas de $Hash(i_0, j_0)$ e com todas de $Hash(i_0 \pm 1, j_0 \pm 1)$ (veja a tabela 2.2, que exemplifica o cálculo para uma partícula de coordenadas de grade $(4, 4)$). Essas buscas pelas partículas de interesse podem ser otimizadas via ordenamento da tabela de pares segundo seus códigos de *hash*.

	Para a partícula 1 (i=4,j=4)	
hash(4-1,4+1)= 29	hash(4-0,4+1)= 37	hash(4+1,4+1)= 45
hash(4-1,4+0)= 28	hash(4-0,4+0)= 36	hash(4+1,4+0)= 44
hash(4-1,4-1)= 27	hash(4-0,4-1)= 35	hash(4+1,4-1)= 43

Tabela 2.2: Voxels de interesse ao estudo de colisão com a partícula 1

Partícula	Hash Associado
4	8
2	22
3	36
1	36
5	36
6	39
7	43

Tabela 2.3: Tabela ordenada pelo campo *hash*

O que resultaria no teste de colisão efetiva (*narrow phase*) com as partículas 3 e 5 (de códigos de *hash* iguais a 36), e com a partícula 7 (de código igual a 43), podendo essas serem localizadas a partir da tabela 2.3 em $O(\log n)$ (busca binária) em lugar de $O(n)$ (força bruta). Mas esse esforço computacional pode ser diminuído pelo uso efetivo da grade através emprego dos campos "início" (I) e "fim"(F) por voxel (V). Esses campos são preenchidos com o primeiro e o último índice (posição) na tabela de pares (após ordenação), que ocuparem um mesmo voxel (tiverem códigos de *hash* iguais). Na tabela 2.4, é apresentado o preenchimento dos campos I e F (representados como *arrays* distintos I[ ] e F[ ]) para o exemplo 2D (veja Fig. 2.4), e, na tabela 2.5, é apresentado um exemplo de busca de candidatos à colisão com a partícula 3 usando os campos I e F (também representados como *arrays*).

i (índice)	0	1	2	3	4	5	6
Hash[i]	8	22	36	36	36	39	43
Partículas[i]	4	2	3	1	5	6	7
I[Hash[i]]	0	1	2			5	6
F[Hash[i]]	0	1			4	5	6

Tabela 2.4: Ocupação dos voxels

Anterior ao passo de preenchimento dos campos de "início" e "fim", todos os voxels deverão ser inicializados com uma indicação de "vazio", por exemplo, atribuindo-se -1 aos seus campos I. Com essa estrutura volumétrica, em lugar de uma busca binária para se encontrar uma determinada partícula, um simples cálculo aritmético ($O(1)$) será suficiente para a localização das partículas candidatas à colisão.

Partícula "3"	Números de hash a serem pesquisados	Ocupação	Partículas com as quais a "3" será testada
Hash["3"]=36	27,28,29, 35,36,37, 43,44,45,	I[27]=-1, F[27]=... I[28]=-1, F[28]=... I[29]=-1, F[29]=... I[35]=-1, F[35]=... I[36]= 2, F[36]= 4 I[37]=-1, F[37]=... I[43]= 6, F[43]= 6 I[44]=-1, F[44]=... I[45]=-1, F[45]=...	 Partícula[2]= "2" Partícula[4]= "1" Partícula[6]= "5"

Tabela 2.5: Exemplo de uso dos campos I e F dos voxels para a localização eficiente das partículas.

No *pipeline* de uma simulação baseada em partículas (veja Fig.2.11), uma boa parte da fração do tempo de cada iteração costuma ser dedicada à construção da estrutura de grades, especificamente, ao ordenamento de pares [24], sendo, portanto, um dos principais gargalos desse tipo de animação. Segue-se o algoritmo básico da *construção da estrutura de grades*:

Algoritmo 2.1: Construção da estrutura de grades

```
// Construção da tabela de pares e ordenação
for i=0 to nParticles-1
  Index[i]=i;
for i=0 to nParticles-1
  Hash[i]=HashFunc(X[i].x, X[i].y, X[i].z);
SortPairs(Hash[],Index[],nParticles);
// limpeza da grade
for i=0 to nCells-1
  I[i]=-1;
```

```
// realização da correspondência entre célula e array ordenado
for i=0 to nParticles-1 {
  thisHash=Hash[i];
  iFirst=iLast=i;
  if (i!=0) { if (thisHash==Hash[i-1]) iFirst=-1; }
  if (iFirst!=-1) I[thisHash]=iFirst;
  if (i!=(nParticles-1)){ if (thisHash==Hash[i+1]) iLast=-1; }
  if (iLast!=-1) F[thisHash]=iLast;
}
```

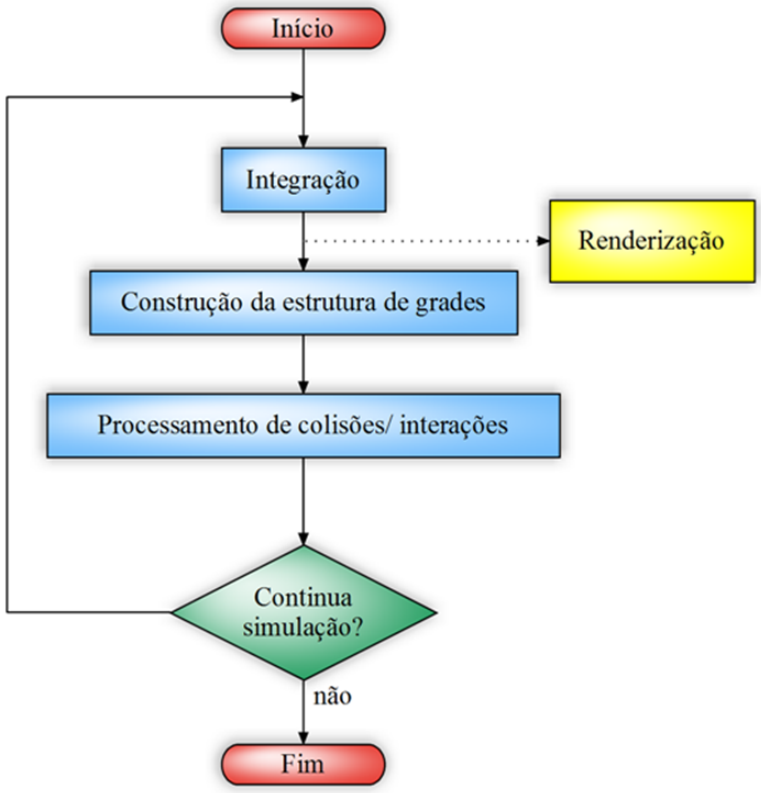

Figura 2.11: *Pipeline* genérico de uma simulação de um sistema de partículas

2.6 Computação paralela e a unidade de processamento gráfico

É fato que simulações de sistemas de partículas requerem grande poder computacional, principalmente, quando se objetiva animações a taxas interativas com alto grau de realismo. Num passado não muito distante, o aumento do poder computacional era quase que exclusivamente vinculado à adequação dos processadores para trabalharem com frequências (*clocks*) mais altas. Contudo, em função de problemas técnicos inerentes à tecnologia de circuitos integrados [4], as possibilidades de continuidade de aumento

de *clock* foram bem reduzidas.

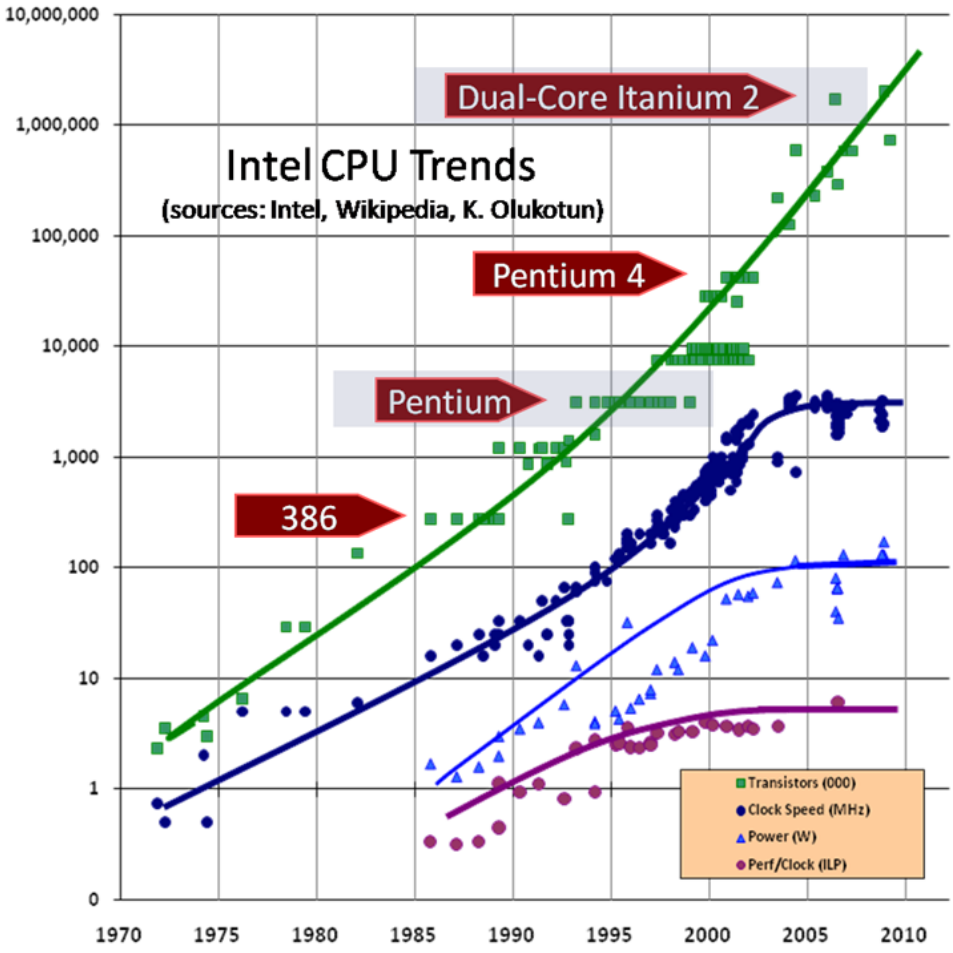

Figura 2.12: Evolução das CPUs (figua obtida de [4])

Por essa estagnação de desempenho (veja Fig. 2.12), o paralelismo, antes confinado a arquiteturas especializadas usadas por cientistas e engenheiros, [25] começou a receber atenção dos maiores fabricantes de processadores destinados a PCs: instruções de manipulação de múltiplos dados (e.g., extensões SSEx da Intel); soluções em virtualização de processadores e, é claro, os processadores efetivamente multi-núcleos.

Em [26] é mostrado que apenas os softwares preparados para execução concorrente poderão obter sensível melhora de desempenho com o advento de novas tecnologias (que, na grande maioria dos casos, tem sido na área de processamento paralelo).

Então, um dispositivo que, nos primórdios, era usado apenas para geração de sinal de varredura para CRTs [27] é, na atualidade, um dos principais aliados nas soluções *many-core* utilizadas na ciência e no entretenimento. Esse dispositivo é conhecido como "placa de vídeo", ou simplesmente, GPU.

As primeiras placas de vídeo a processarem algum tipo de geometria (tarefa antes restrita à CPU) começaram a surgir por volta de 1983. Elas contavam somente com aceleração 2D para desenho de linhas, retângulos, arcos e caracteres baseados em bit-

maps. Apenas na década de 90, placas gráficas com aceleração 3D foram concebidas e trazidas ao mercado consumidor. Elas permitiam a transformação de vértices e cálculo de iluminação via hardware, mas segundo um pipeline fixo [28]. Em 2001, surge a primeira placa gráfica com a possibilidade de rodar pequenos programas não nativos no hardware para manipulação de vértices e fragmentos, ou seja, o pipeline fixo havia se tornado uma opção. Esses programas ficaram conhecidos a partir de então como *shaders*. Ano após ano, pela evolução das GPUs, a programação dos shaders rapidamente ganhou flexibilidade na direção da programação de propósito geral das CPUs (programas maiores, mais complexos, laços de repetição etc). Isso não trouxe apenas mais realismo na apresentação de gráficos, mas também trouxe a super computação ao alcance dos PCs, tornando-se um grande propulsor nas áreas de matemática, física e engenharia [29].

A popularidade das GPUs como processadores de uso geral não é vinculada exclusivamente à grande velocidade de processamento e ao paralelismo de "baixo" custo, mas também às ofertas de ferramentas para esse fim, como CUDA, OpenCL, Brook, CTM, etc [30]. Tais ferramentas permitiram abstrair a complexidade de mapear conceitos tradicionais de programação para conceitos de computação gráfica necessários ao eficiente emprego das GPUs[31].

2.7 *Sorters* usados como referência neste trabalho

Nas simulações baseadas em partículas usando GPU, duas sugestões de ordenamento de *arrays* se tornaram bastante populares[32]: *bitonic sort* e *radix sort*. Isso se deveu a eficiência das implementações e, é claro, à gentil disponibilização dos códigos fontes no SDK da NVIDIA [33]. Os dois algoritmos serão descritos a seguir:

2.7.1 *Bitonic Sort*

O algoritmo do *bitonic sort* foi apresentado por Ken Batcher em 1968 [34]. Seu funcionamento consiste em criar subsequências bitônicas (de 4 itens de comprimento, inicialmente) a partir de um array arbitrário (comprimento potência de 2), que serão fundidas, dando origem a sequências bitônicas com o dobro do tamanho. Procedendo dessa forma, após um total de $\frac{\log n*(\log n+1)}{2}$ subpassos, a sequência inicial estará ordenada. Na figura 2.13, são apresentados 3 exemplos da aplicação desse algoritmo.

Embora sua complexidade seja $O(n \log^2(n))$, em vários contextos, consegue ser mais rápido que seus concorrentes de menor complexidade [35]. Uma de suas características mais interessantes é que a sequência de comparações independe da disposição dos dados. Esta característica o torna bastante suscetível, inclusive, à implementação em hardware especializado [36]. Outras características se destacam nesse método: ordenamento sem necessidade de buffer auxiliar e não ser estável (não permanência de ordem inicial relativa

entre chaves de mesmo valor, veja Fig. 2.14). Na sua forma inicial, ele se aplica apenas às listas de comprimento potência de dois, mas, com adaptações, pode ser aplicado a comprimentos arbitrários também [37].

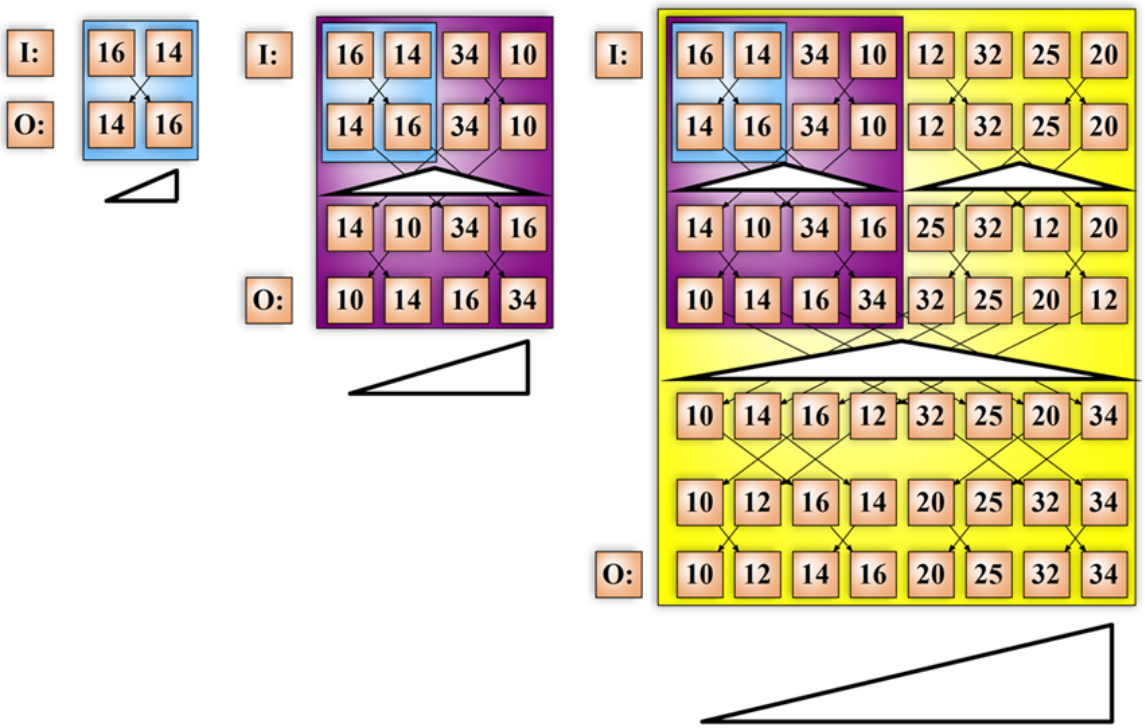

Figura 2.13: Ilustração do funcionamento em 3 sequências (a primeira não contou com nenhuma sequência bitônica)

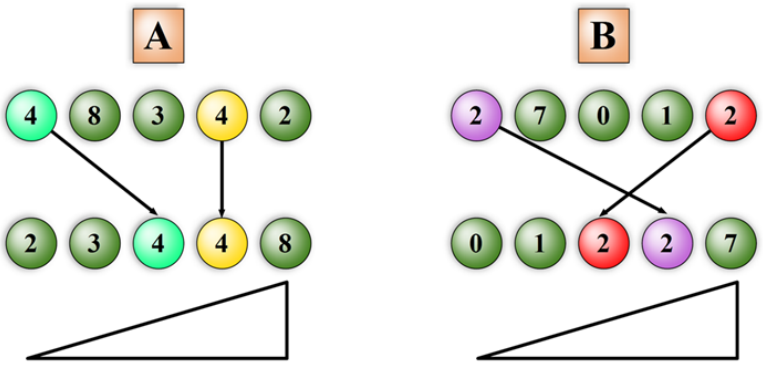

Figura 2.14: Em “A”, foi aplicado um procedimento estável de ordenação. Em “B”, um não estável.

A versão disponibilizada no SDK da NVIDIA, visando à eficiência, aproveita o fato de muitas subsequências, em razão de seus comprimentos, poderem ser tratadas em memórias de acesso rápido, isso devido às trocas de posição de itens próximos.

2.7.2 *Radix Sort*

Historicamente, uma proposta similar ao *radix* teve seu primeiro emprego registrado em 1890 na realização de um censo populacional. A idealização do dispositivo que o permitiu é creditada ao estatístico Herman Hollerith. Na computação, foi apresentado por Harold H. Seward em 1954 para ordenamento de cartões perfurados [38]. Em lugar de ser um método baseado na comparação de itens, ele tem seu ordenamento baseado no uso direto dos dígitos componentes das chaves, enquadrando-se então como um método baseado em distribuição [39]. Na forma original, ele opera sobre diversos tipos de dados, mas todos baseados em tipos inteiros. Mas com pequenas modificações no arranjo dos bits que representam os dados, é possível cobrir outros tipos. Atualmente o termo *radix* está mais associado a uma classe de métodos do que a uma solução específica baseada em dígitos, sendo então classificadas em dois tipos: LSD (Least Significant Digit) e a MSD (Most Significant Digit) [40].

De forma geral, tanto o radix LSD quanto o MSD processam os elementos em etapas que permutam os elementos de forma a agrupar os itens da lista segundo um determinado dígito (relativo a etapa). Esse dígito, no caso especificamente computacional, é formado por um grupo de bits consecutivos.

O algoritmo *radix* MSD começa a percorrer os dígitos a partir do mais significativo até o menos significativo. O *radix* LSD procede de maneira inversa. Mas essas diferenças trazem implicações que vão além da simples opção de sentido na visita aos dígitos.

A variante *radix* LSD opera separando as chaves em *buckets* segundo o dígito (da etapa) mantendo a ordem relativa entre os itens, e, depois, os devolve a uma única tabela, que sofrerá o mesmo processo para o próximo dígito (exemplo na Fig. 2.15). Uma decorrência interessante desse funcionamento é que esse método requer um procedimento de ordenação estável por etapa (*counting sort*, por exemplo) para que, de fato, funcione como um procedimento de ordenação como um todo.

O *radix* MSD opera de forma similar ao LSD, mas em lugar de devolver a chave para uma tabela que será tratada como única no passo seguinte, cada *bucket* (ou os dados que o representam) será(ão) tratado(s) como uma tabela disjunta das outras, que sofrerá então um *radix* MSD para o próximo dígito, ou seja, um procedimento recursivo. Interessante também é que, por começar da esquerda (sequência lexicográfica), ele acaba tendo uma vocação natural para ordenar strings, que em essência são chaves de tamanhos diferentes (exemplo na Fig. 2.16).

O método *radix* utilizado nesse trabalho é baseado no trabalho de Satish [41], que é uma versão LSD que utiliza 4 bits por passo (em lugar de 1 bit do *radix* proposto por Blelloch[42]). Além disso, o código para essa dissertação teve uma ligeira modificação para comportar ordenação de pares (32 e 32 bits) a exemplo do *bitonic sort* mencionado anteriormente.

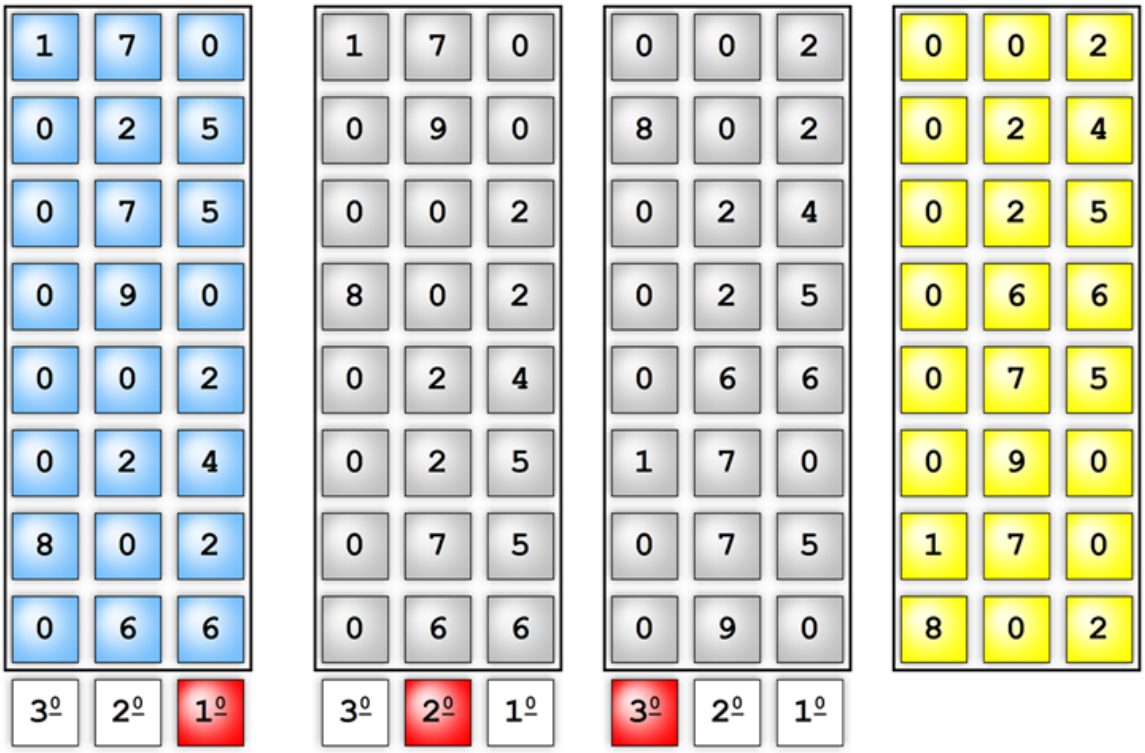

Figura 2.15: Exemplo do radix LSD

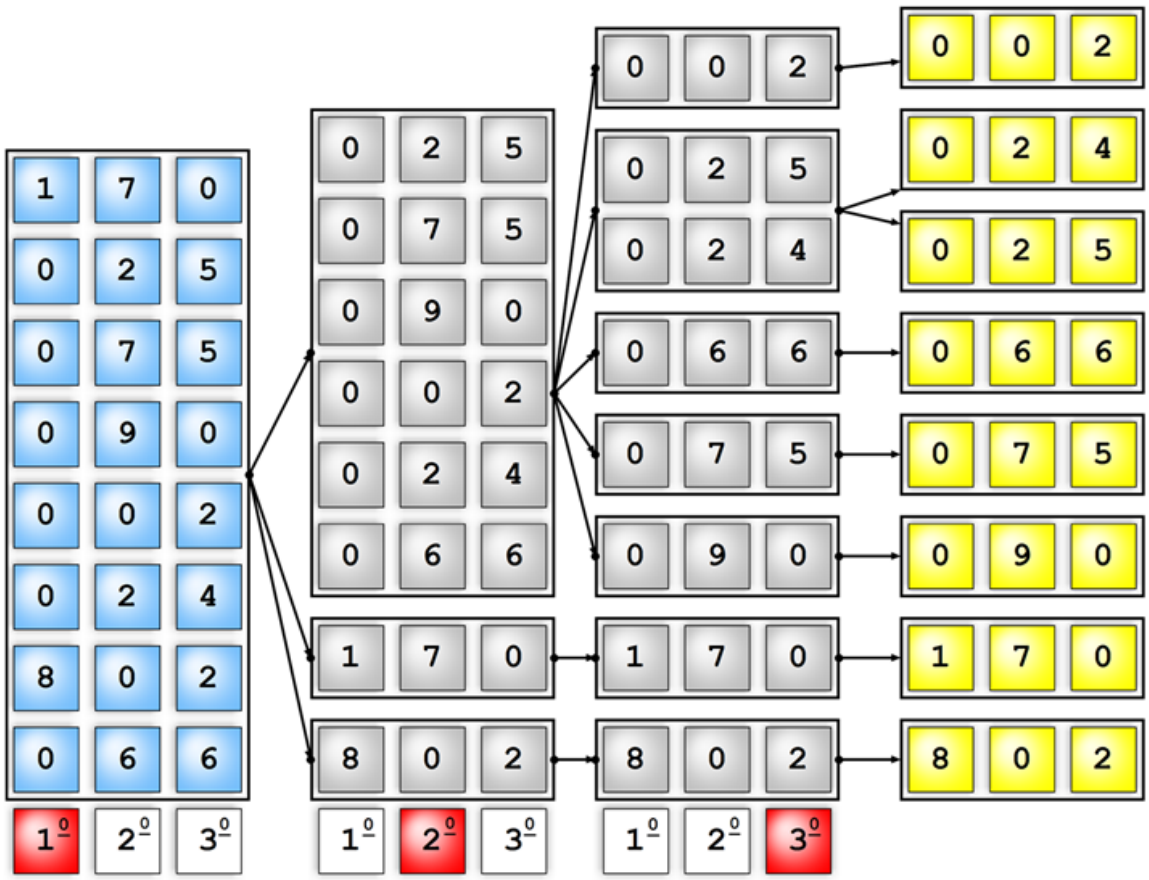

Figura 2.16: Exemplo do radix MSD

2.8 Sensibilidade dos métodos *Bitonic Sort* e *Radix Sort* em relação a tabelas parcialmente ordenadas

Em várias ocasiões práticas, quando em soluções de ordenação sequencial, fica evidente que o ordenamento de tabelas parcialmente ordenadas é mais rápido que o ordenamento daquelas completamente desordenadas. Isso se deve geralmente ao menor número de trocas requeridas para a tarefa. Em [43], são apresentadas noções gerais de métodos adaptativos que buscam obter melhor desempenho através desses cenários. Seria então algum desses dois eficientes métodos de ordenação, que não são métodos adaptativos, sensivelmente mais rápidos quando atuando sobre tabelas parcialmente ordenadas? Se sim, como apresentar tabelas nessa disposição aos *sorters* a partir das simulações envolvendo partículas? Esse será o tópico norteador do próximo capítulo.

Capítulo 3

Ordenamento otimizado de partículas

Neste capítulo, são apresentados os fundamentos e as técnicas que possibilitaram a aceleração no processo de ordenamento de pares.

3.1 Coerência espacial-temporal e tabela de pares parcialmente ordenada

Um dos aspectos mais importantes em animação é a suavidade na transição de quadros. Essa "suavidade" é traduzida por similaridades entre estados consecutivos no tempo (veja o exemplo na fotografia 3.1). Tal similaridade entre estados é melhor referenciada por coerência.

Figura 3.1: Continuidade no movimento (figura obtida em [5])

Em [44] vários tipos de coerências são enumerados e bem descritos. Muitos desses tipos, pela própria natureza de inter-relação entre coerências, possuem definições que acabam por abranger parcialmente ou integralmente outros tipos. Dessa forma, ao longo do

texto, optou-se por “coerência temporal e espacial” para sintetizar onde se fundamentou a proposta de aceleração das simulações envolvendo partículas . Em computação gráfica, a coerência tem encontrado aplicação em compactação de dados (vídeos e imagens) [45], estabilização de vídeos [46]; e, também, na “compactação” de tarefas, ou seja, conferir mais velocidade aos processos [47].

Mas como acelerar as animações baseadas em partículas usando essas coerências?

3.2 Tentando aproveitar a tabela parcialmente ordenada do quadro

Numa simulação suave, empregando grades uniformes para subdivisão do espaço, espera-se que muitas partículas tendam a permanecer numa mesma célula no quadro seguinte. Essa permanência, nos algoritmos baseados em ordenação, sugere a ideia de que a tabela de pares (*hash*-índice) se mantenha com muitos itens inalterados em relação à tabela de pares do quadro anterior. Ou seja, uma parte dela (tabela) se encontra mantida (e ordenada) e a outra parte da tabela se encontra modificada (e, provavelmente, não ordenada). As figuras 3.2 e 3.3 exemplificam a consecução de 3 quadros e suas decorrências na tabela de pares ordenada segundo o algoritmo tradicional apresentado na seção 2.5.

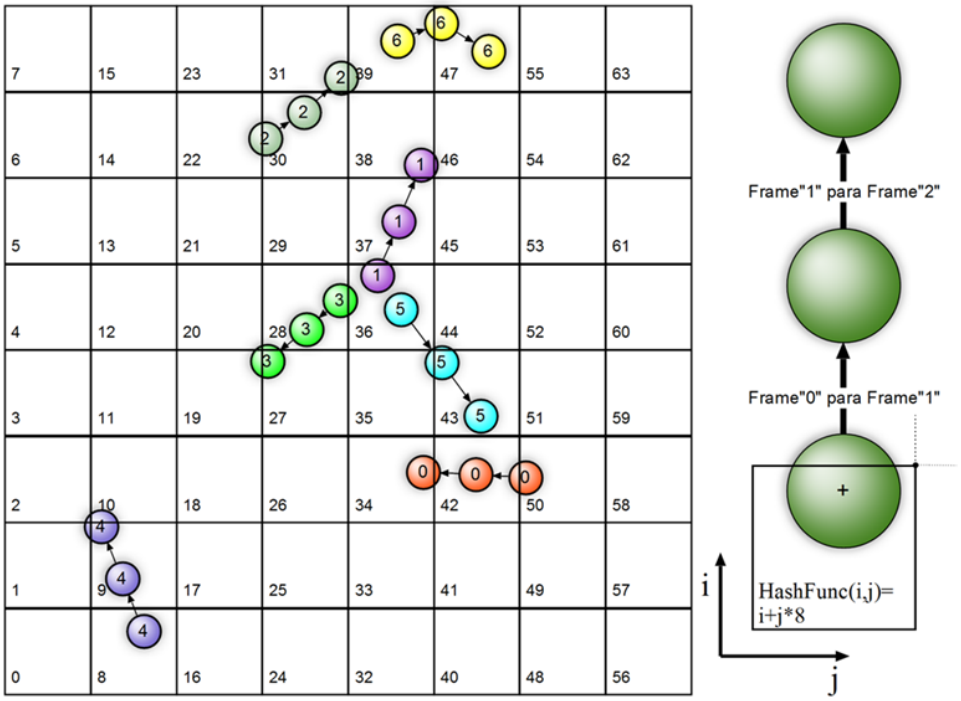

Figura 3.2: Exemplo de 3 frames consecutivos

Como se pode observar a partir da figura 3.3, a sucessão de quadros teve como implicação a grande similaridade entre tabelas de pares ordenadas, o que motivou a primeira tentativa de aproveitamento de coerência, que consistiu em fazer a atualização dos

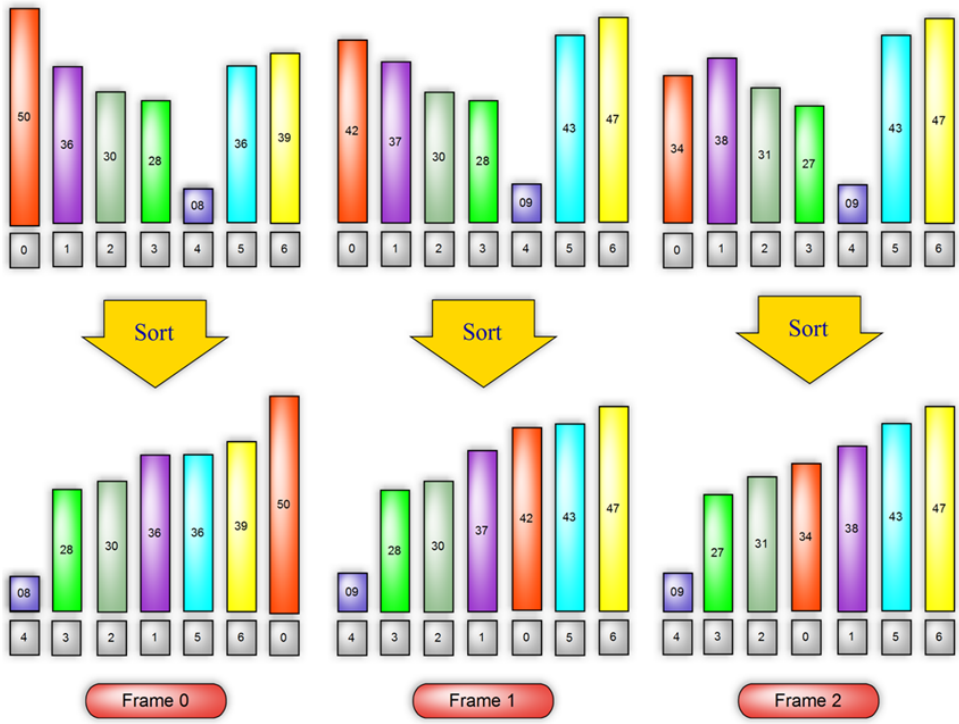

Figura 3.3: Exemplo de ordenamentos de pares de 3 frames consecutivos

valores de *hash* a partir da sequência da tabela de pares da última ordenação (algoritmo 3.1). Isto é conseguido pelo emprego de mais um nível de indireção no acesso aos pares. Em particular, define-se uma tabela auxiliar `Index`, que é inicializada com os índices correspondentes à ordenação anterior, conforme o algoritmo 2.1.

Algoritmo 3.1: Construção da tabela de pares usando a ordenação anterior

```
for i=0 to nParticles-1
  Index[i]=i;
Label1:
// Construção da estrutura de grades
for i=0 to nParticles-1 {
  j=Index[i];
  Hash[i]=HashFunc(X[j].x, X[j].y, X[j].z);
}
SortPairs(Hash[],Index[],nParticles);
...
if (flagContinueSimulation) goto Label1;
```

Que aplicado sobre a mesma simulação (Fig. 3.2), resultou na nova configuração de pares conforme a figura 3.4.

Nesse novo cenário, testes de desempenho dos *sorters* foram conduzidos, mas não foi observada nenhuma alteração significativa de desempenho associada à nova distribuição de pares. Na seção 3.3, serão apresentados os métodos utilizados nessa avaliação junta-

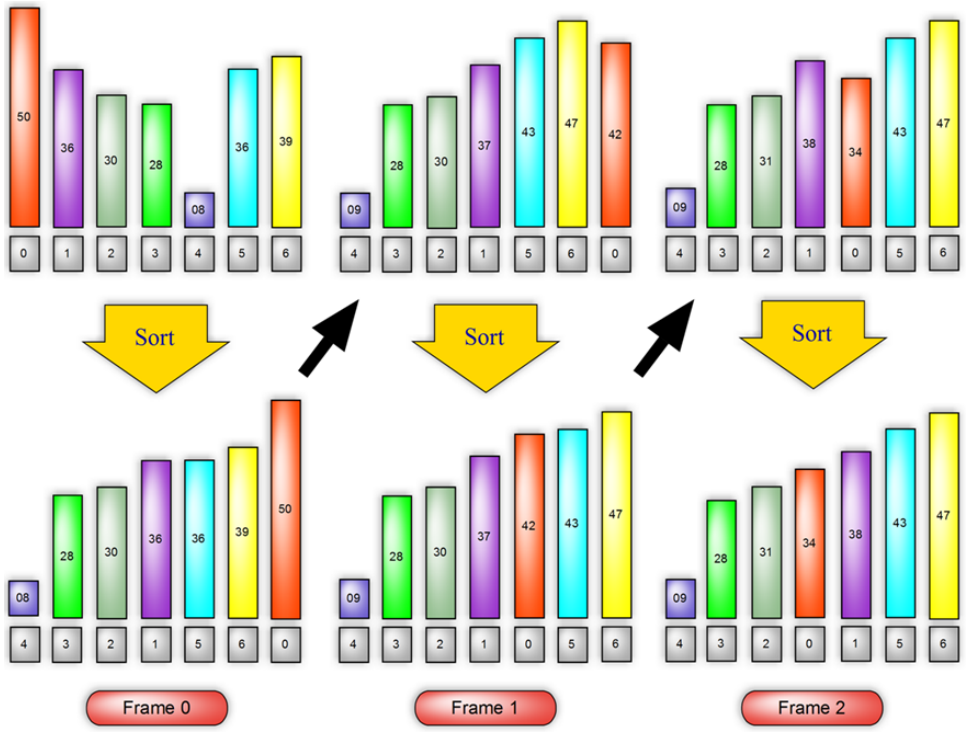

Figura 3.4: Exemplo de ordenamentos de pares de 3 frames consecutivos pelo algoritmo modificado

mente com os testes de desempenho perante tabelas parcialmente ordenadas.

3.3 Sensibilidade dos métodos *bitonic sort* e *radix sort* em relação a tabelas parcialmente ordenadas

Seria então algum desses dois eficientes métodos presentes no SDK da NVIDIA para processamento em GPU, que não são métodos adaptativos, sensivelmente mais rápidos quando atuando sobre tabelas parcialmente ordenadas? Ou seja, haveria um aproveitamento da coerência temporal e espacial inerente ao código nativo dessas implementações?

Para responder a essa pergunta, foi elaborado um conjunto de experimentos que objetivou a obtenção dos tempos médios de ordenamento para 50 amostras; que foram avaliados por faixa (16k a 1M pares, potência de 2), por método (*bitonic sort* e *radix sort*[1]) e por perturbação (1% a 99% de alteração, passo de 2%). As chaves foram de 20 bits, o que favoreceu o método *radix* sem fugir de um cenário real (e.g. adoção de uma grade $64 \times 64 \times 64$, usando 6 bits por eixo). No *bitonic sort*, o tamanho das chaves não é parâmetro que altere a quantidade de passos requeridos.

Após testes realizados no Sistema 1 (descrição em D.1), nenhuma grande tendência de melhora foi observada nos tempos médios levantados (gráficos em A.1 e A.2). Rati-

[1] A implementação desse radix não cobre faixas inferiores a 32k pares.

ficando o experimento, em [48], o *radix sort* já era qualificado como fraco quanto à melhora de desempenho perante entradas parcialmente ordenadas. O método de produção dos parâmetros de prova consistiu basicamente em montar uma tabela de pares com chaves aleatoriamente geradas, que sobre si (tabela), após sua ordenação, fora aplicada a perturbação conforme o exemplo abaixo. O método utilizou função rand() de geração de números *pseudo aleatórios* inteiros da biblioteca *stdlib* (MSVC 2008).

Algoritmo 3.2: Exemplo de aplicação de 30% de perturbação nas tabela de chaves

```
// Exemplo para aplicação de 30\% de alteração:
// randf() <-> uma versão normalizada de rand()
// VMAX <-> ( 1<<(3*6) ) -1
for i=0 to TotalElementos-1
  val[i]=VMAX*randf();
Sort(val[],TotalElementos);
for i=0 to TotalElementos-1
  val[i]=( randf() <= 30% ) ? val[i] : VMAX*randf();
```

É interessante observar que, embora faça sentido a ideia de que a ordenação de tabelas parcialmente ordenadas seja sempre mais rápida que a ordenação de uma completamente desordenada, a "prática computacional" revela muitas vezes a indiferença de desempenho e, em alguns casos, até a piora (algumas implementações do *quick sort* [49]).

As sugestões que efetivamente aproveitarão os *arrays* parcialmente ordenados em quadros sucessivos serão apresentadas a seguir.

3.4 Explorando a coerência temporal e espacial (execução sequencial)

A solução a seguir baseou-se em muito na abordagem anterior, mas em lugar de ordenar uma tabela de mesmo tamanho um pouco mais organizada, essa solução foi balizada pela simples idéia de que ordenar mais rápido seria ordenar uma tabela menor.

Observando a sucessão da tabela de pares através da figura 3.4, não é difícil observar que a atualização do *array* de códigos de *hash* demonstra uma grande similaridade entre tabela ordenada anterior e a tabela a ser ordenada (parte superior direita da anteriormente ordenada); isso traduz a ideia de que "uma parte da tabela se encontra mantida (e ordenada) e outra parte da tabela se encontra modificada (e provavelmente não ordenada)"; o que sugere a seguinte modificação no algoritmo anterior:

1. Montar uma nova tabela fazendo uso da última ordenação (a exemplo do que foi feito na seção 3.2). Veja figura 3.5.

   ```
   for i=0 to nParticles-1 {
   ```

```
  j= Index[i];
  Hash[i]=HashFunc(X[j].x, X[j].y, X[j].z);
  }
```

2. Por comparação entre a última tabela ordenada e a gerada pelo passo 1, serão geradas duas tabelas: uma de itens mantidos e outra de modificados (Fig. 3.6). Observa-se que a partir deste passo até o passo 4 inclusive, todo o procedimento foi sequencial.

```
// Split da tabela de pares
nMod=nKept=0;
for i=0 to nParticles-1 {
  if (HashOld[i]!=Hash[i]) {
    modifiedHash[nMod]=Hash[i];
    modifiedIndex[nMod]=Index[i];
    nMod++;
  } else {
    keptHash[nKept]=Hash[i];
    keptIndex[nKept]=Index[i];
    nKept++;
  }
}
```

3. Ordenar a tabela de itens modificados (Fig. 3.7).

```
SortPairs(modifiedHash[],modifiedIndex[],nMod);
```

4. Intercalar as tabelas dos itens mantidos e dos itens "não mantidos" ordenados , resultando na tabela ordenada desse frame (Fig. 3.8).

```
MergePairs(Hash[],Index[], // <-- arrays de resposta
           keptHash[],keptIndex[],nKept,
           modifiedHash[],modifiedIndex[],nMod);
```

Na contramão das atuais tendências, essa simples implementação fazendo uso do *quick sort* na ordenação dos modificados conseguiu tornar esse procedimento bastante competitivo inclusive perante implementações bastante eficientes de ordenamento de pares em GPU (resultados no capítulo 4).

A complexidade do procedimento (como custo temporal T) continuou a mesma[2], mas a parcela mais custosa (ordenação) foi substituída em parte por uma tarefa bem mais veloz (e de menor complexidade), vindo daí a melhora do desempenho. Em outras palavras, se das n partículas, m têm seus códigos modificados, então

[2]Assumindo o pior dos casos, que ocorre quando o ordenamento integral é requerido ($m = n$).

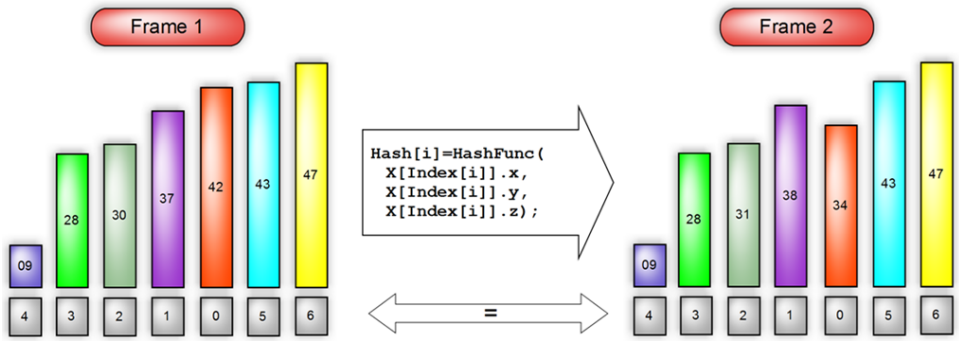

Figura 3.5: Programação da tabela de *hash* via idexação pela última ordenação

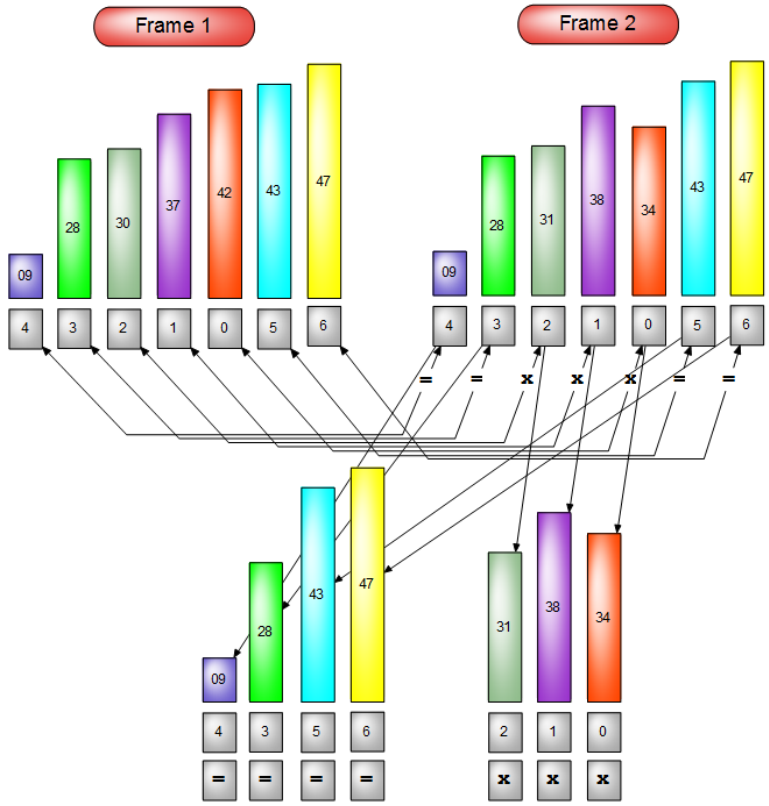

Figura 3.6: Comparação e *split* da tabela em "mantidos" e "não mantidos"

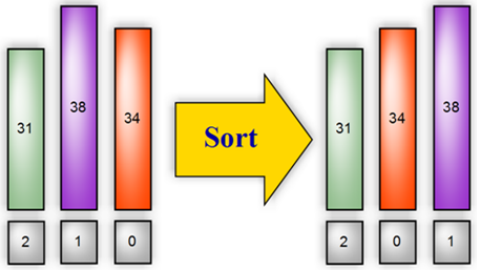

Figura 3.7: Ordenação da tabela de "não mantidos"

$$T = O_{split}(n) + O_{merge}(n) + O_{sorter}(m).$$

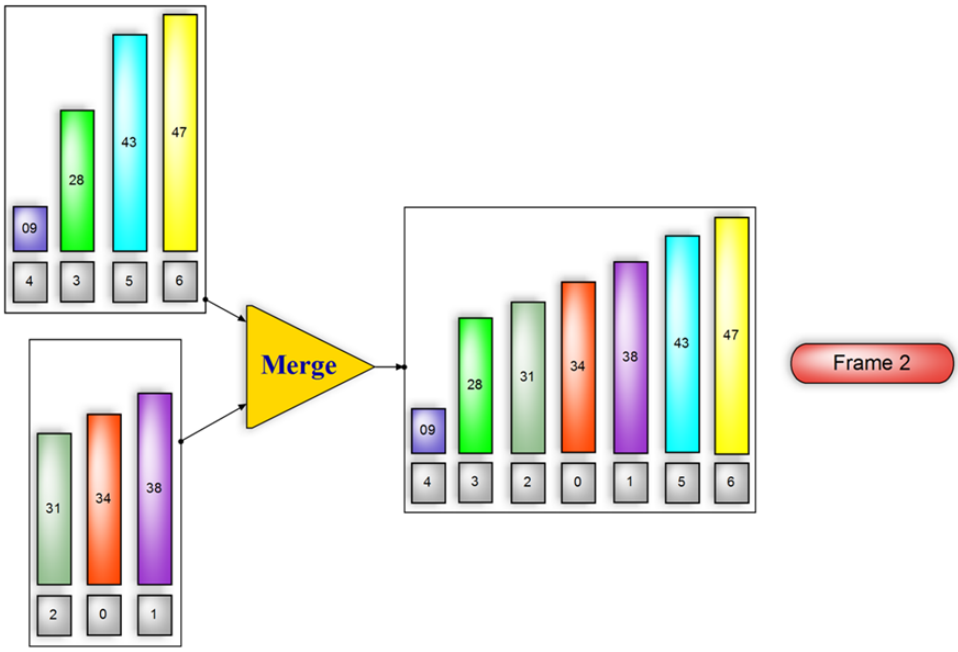

Figura 3.8: *Merge* das tabelas

3.5 Explorando a coerência temporal e espacial (execução em paralelo)

São duas as abordagens sugeridas: uma bastante similar à solução sequencial e outra usando operações atômicas.

3.5.1 *Split* e *Merge* em GPU

À semelhança da solução sequencial apresentada na seção 3.4, os algoritmos *split* e o *merge* integram esse procedimento, ficando então a diferença por conta da implementação, que é destinada ao paralelismo oferecido pela GPU.

Mas como fazer tais métodos "tão sequenciais" desfrutarem do paralelismo? O *split* paralelizado é baseado em [42], que faz uso do método *scan* disponibilizado no SDK da NVIDIA para OpenCL. Vale ressaltar que essa mesma solução de *scan* foi a utilizada na implementação do *radix sort* empregado neste trabalho. O método adotado para ordenamento das tabelas de pares modificados foi uma composição dos métodos *radix sort* e *bitonic sort*, onde o mais rápido é escolhido (veja Fig. 3.9). Este processo de escolha é função essencialmente do comprimento do *array* e do número de bits empregados como chave, conforme detalhado no capítulo 4.

O *merge* foi baseado no trabalho de Ottmann [50]. Seu funcionamento consiste basicamente no ranqueamento dos itens seguido pela cópia dos dados diretamente no *array* de resposta. O *rank* de um dado nada mais é que a contagem do número de itens menores que ele num determinado *array*. Quando esse *array* se encontra ordenado, a obtenção do

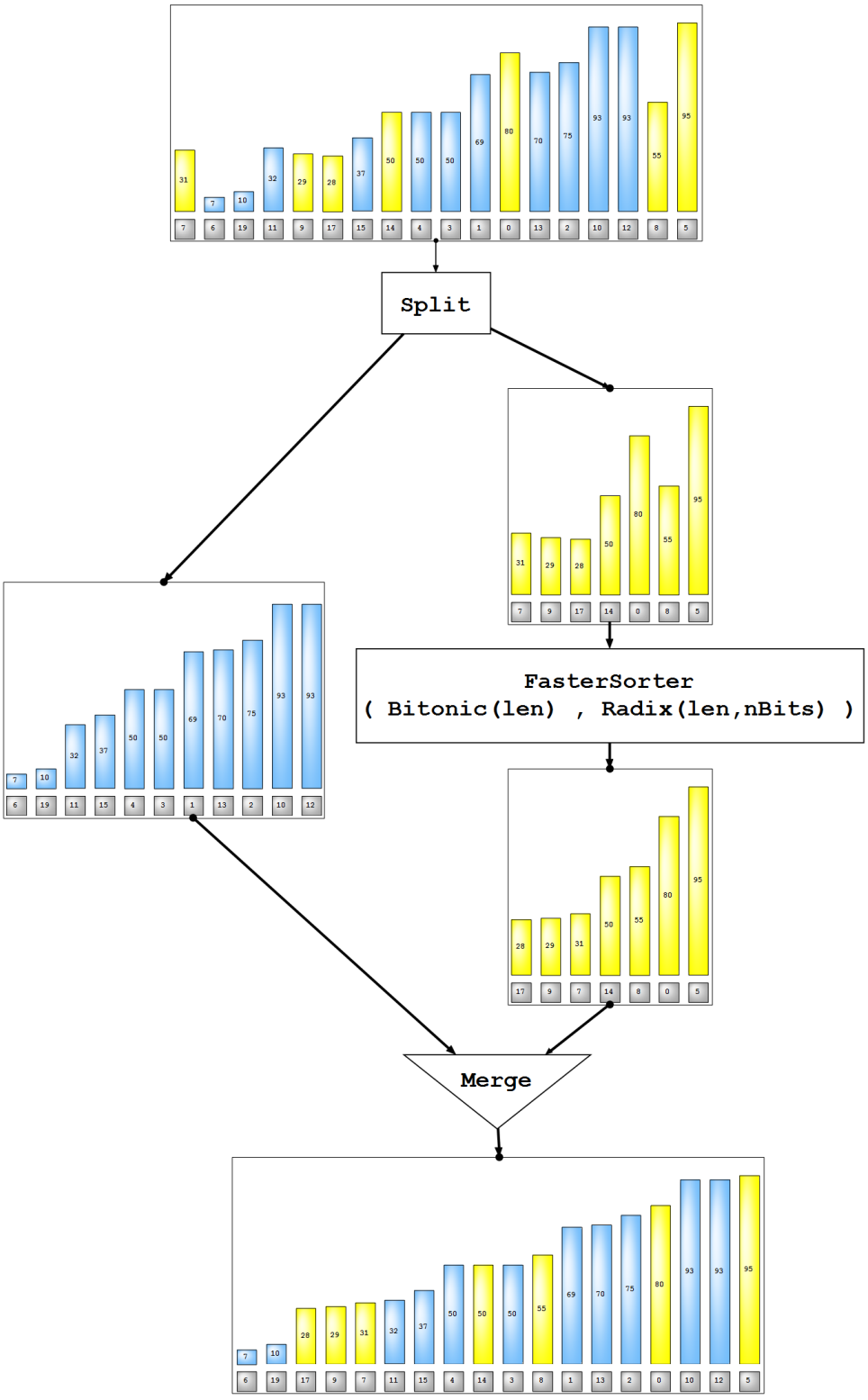

Figura 3.9: Pipeline da ordenação de pares da primeira sugestão para GPU

rank associado a um determinado dado é bastante rápida (via busca binária). O funcionamento desse *merge* confere ao método como um todo a não estabilidade (veja Fig. 2.14). Isso se deve à forma de obtenção do *rank* (em igualdade de chaves, busca-se o extremo maior ou menor).

3.5.2 Usando operações atômicas

A sequência desta solução é um pouco diferente das anteriormente propostas (seção 3.4 e subseção 3.5.1). Aqui, o processo de ordenação ocorre depois do *merge*. Outra característica que a diferencia da primeira sugestão para GPU é o uso de operações atômicas. Uma operação atômica é aquela que, além de ser executada sem interrupção, tem exclusividade no acesso a determinados recursos computacionais, como memória, por exemplo [51]. A ideia é que acessos concorrentes a esses recursos sejam serializados, porém de forma eficiente. Observa-se que, embora se garanta acessos não simultâneos, não existe prioridade, ou seja, não há uma ordem pré-estabelecida entre os acessos.

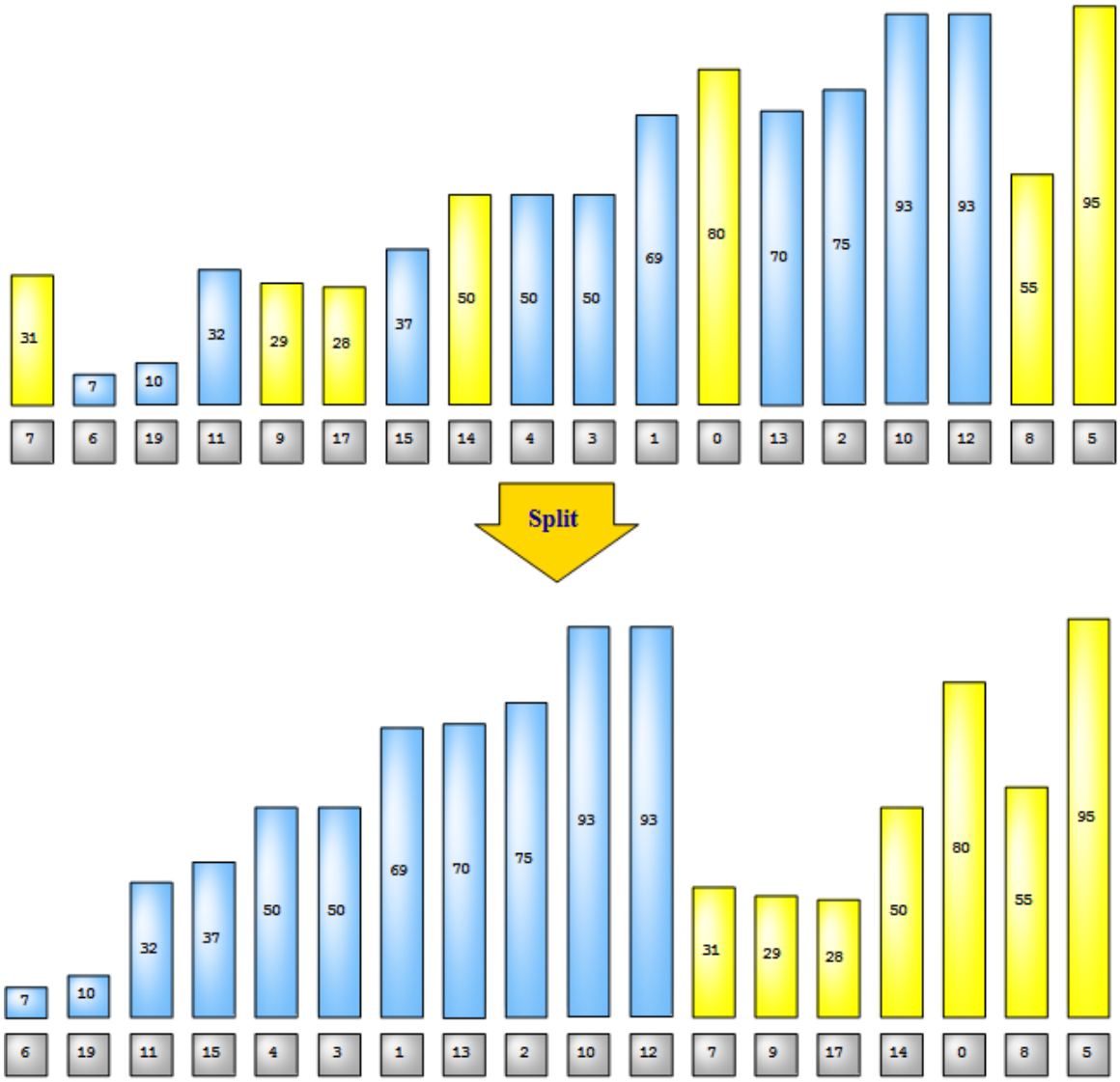

Figura 3.10: *Split* dos pares

Esta solução emprega as mesmas operações da sugestão anterior (subseção 3.5.1), até o passo de partição (Fig. 3.10). Uma vez separados os pares que mudaram e os que não mudaram de célula (respectivamente, barras amarelas e azuis na figura 3.10), executam-se

os seguintes passos (veja o exemplo da Fig. 3.11):

1. Inicialização da *tabela de incidências* A[ ]:

```
for i=0 to NumParesMantidos-1
  A[i]=1;
```

2. Para cada par modificado, busca-se o índice do par mantido que antecederá o referido par modificado, o que é feito por busca binária e registrado em B[ ] e, nesse mesmo passo, é contabilizado o número de pares a serem introduzidos após um dado par mantido em A[ ] e registrado o deslocamento relativo em C[ ].

```
for j=0 to NumParesModificados-1 {
  B[j]=busca_binaria(TabParesMantidos[],TabParesModificados[i])
  FazerAtomicamente {
    C[i]=A[B[j]];
    A[B[j]]=A[B[j]]+1;
  }
}
```

3. Confecção da *tabela de posicionamento final dos itens mantidos* E[ ] via *scan* da tabela D[ ], que é a tabela A[ ] após o passo de incrementos.

```
E[]=scan(D[]);
```

4. Confecção da *tabela posicionamento dos itens modificados* G [] a partir da *tabela de deslocamentos relativos* C[ ], da *tabela de referências* B[ ] e da *tabela de posicionamento final dos itens mantidos* E[ ]:

```
// a tabela F[] só figurou no algoritmo para facilitar
// a visualização na figura referência
for j=0 to NumParesModificados-1 {
  F[j]=E[B[j]];
  G[j]=F[j]+C[j];
}
```

5. Fazendo uso das tabelas E[ ] e G[ ], todos os itens sofrerão uma etapa de reposicionamento (*scattering*) dos itens mantidos e modificados, o que confere apenas ordenamento parcial da tabela de pares final (veja Fig. 3.12).

```
for i=0 to NumParesMantidos-1
  TabPair[E[i]]=TabKept[j];
for j=0 to  NumParesModificados-1
  TabPair[G[j]]=TabMod[j];
```

6. Ordenação das partes que não possuem ordenamento garantido.

```
for i=0 to NumParesMantidos-1 {
  nItens=D[i]-1;
  i0=E[i]+1;
  SortPairs(TabPair[],i0,nItens); // ordenar na tabela TabPair
                                  // a partir do item i0 (inclusive)
                                  // nItens (número de itens)
}
```

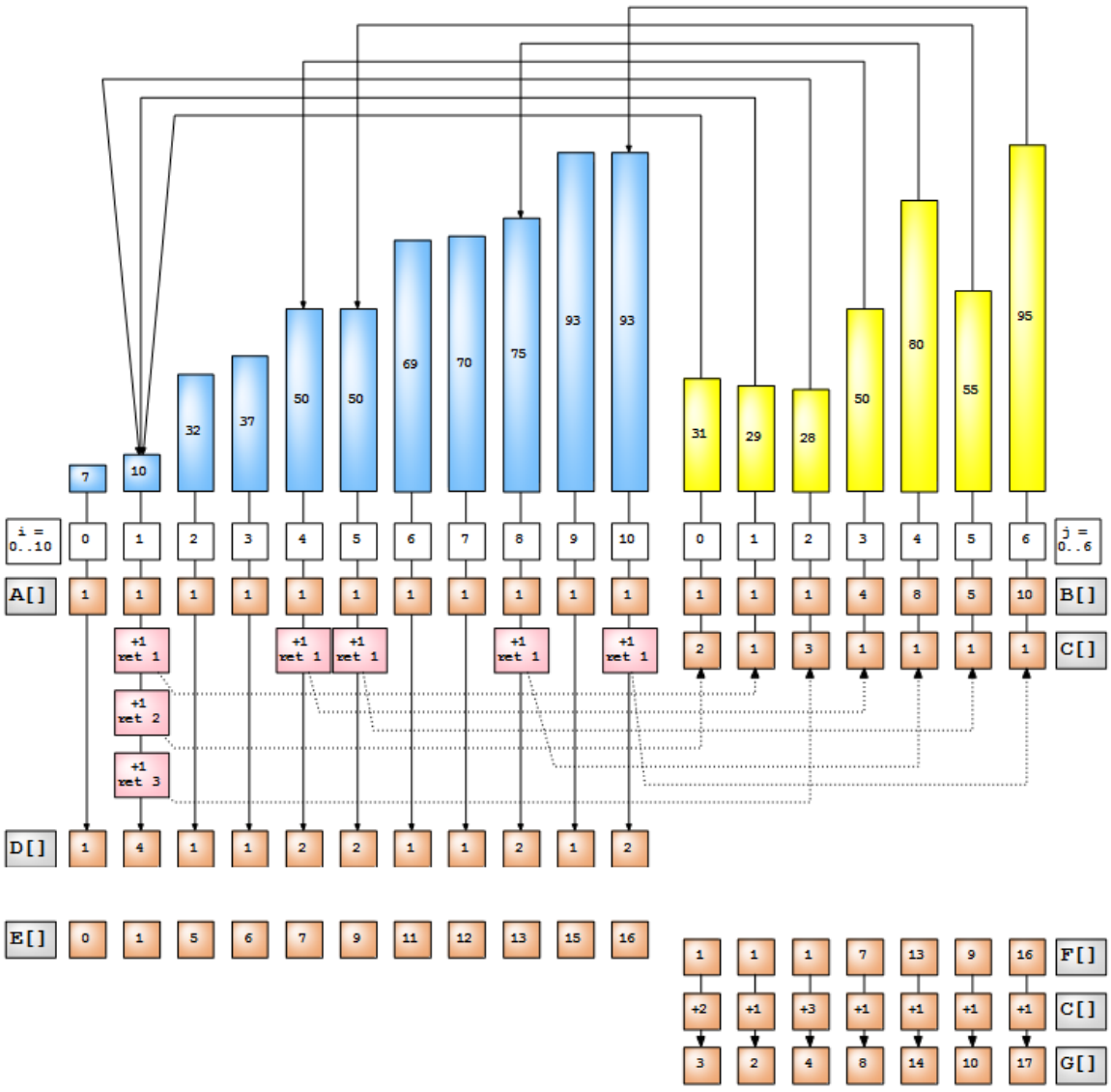

Figura 3.11: Esquema de obtenção de ordenamento parcial

O método de ordenação escolhido para os trechos de inserção foi o *insertion sort*. Essa escolha partiu da hipótese de que poucas inserções vinculadas a um mesmo item (da tabela de mantidos) seriam realizadas. O *bubble sort* também foi testado e teve desempenho

bastante semelhante nos mesmos contextos. Observa-se, também, que esse método de ordenação não é estável (por conta das características das instruções atômicas).

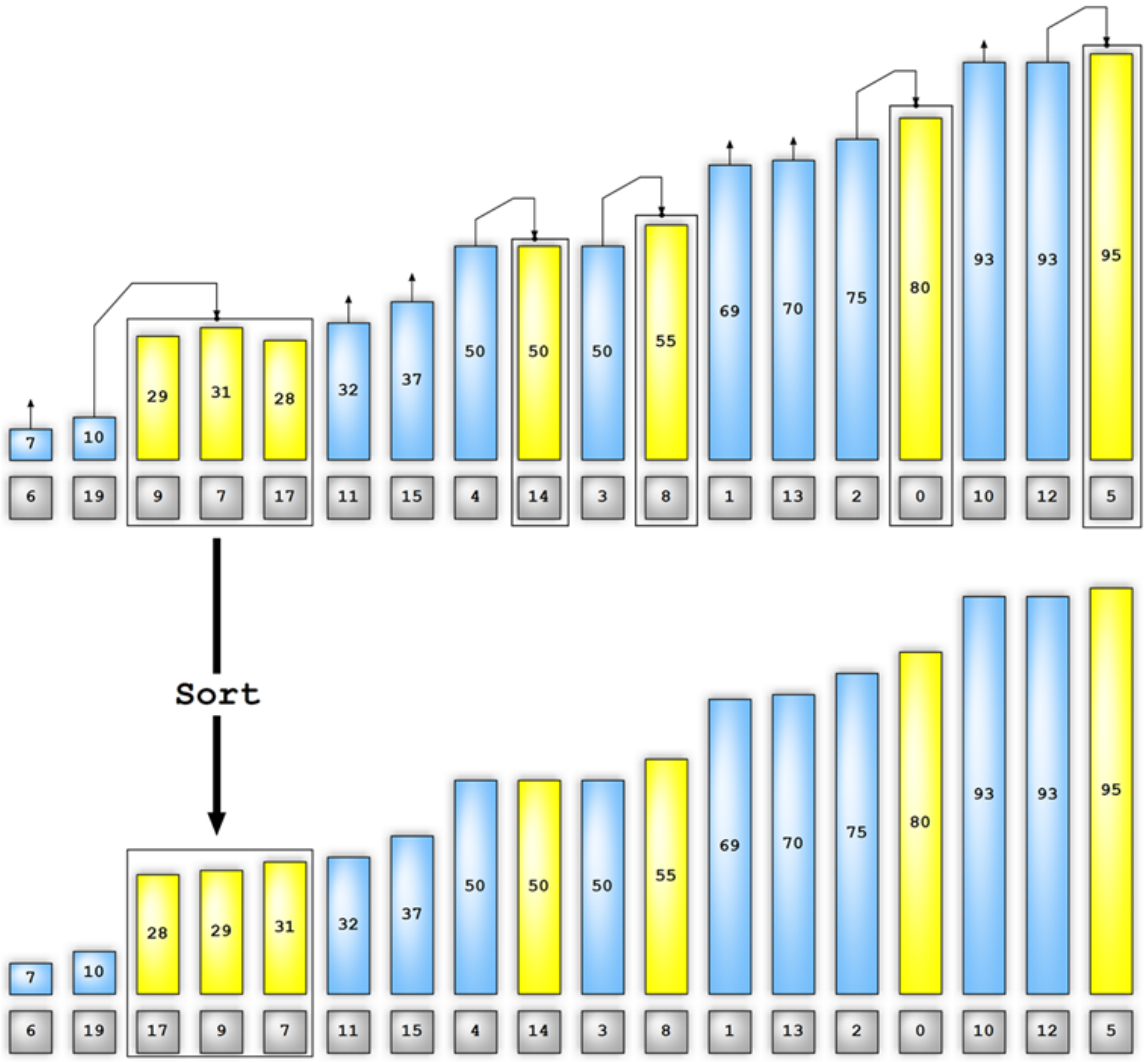

Figura 3.12: Reposicionamento e ordenação

Capítulo 4

Resultados

A avaliação dos métodos apresentados nesta dissertação se dividiu em dois aspectos: quantitativo e qualitativo. O primeiro, que não contou com nenhum tipo de animação ou cálculo de interação física, consistiu apenas na comparação de tempos levados na tarefa de ordenação de *arrays*; o segundo buscou comparar os desempenhos dos métodos em contextos de animações, contando, inclusive, com resposta mecânica, renderização etc.

4.1 Comparação quantitativa

Os resultados de desempenho (*speedups*) são mostrados em detalhes nos gráficos dos anexos B e C. Eles se referem à relação entre os tempos tomados pelos algoritmos de ordenação referência (o mais rápido dentre os apresentados na seção 2.7 para o mesmo comprimento de *array*) e os tempos das versões sugeridas neste trabalho (seção 3.4, subseções 3.5.1 e 3.5.2) conforme a formulação a seguir:

$$Speedup = \frac{Min\left(T_{BitonicSort}, T_{RadixSort}\right)}{T_{avaliado}}.$$

Dessa forma, nos contextos onde o *speedup* de um determinado método for superior a 1.0, o referido se apresentará como mais eficiente perante o algoritmo referência. Uma forma de avaliar o desempenho dos métodos através dos gráficos é observando sua região de desempenho útil, ou seja, a área limitada pelo eixo das abscissas (fração de modificados), pelo eixo das ordenadas (*speedup*) e pela curva de desempenho do método (veja Fig. 4.1). Os pontos indicados nessa figura são os que caracterizam pico de desempenho e menor fração de modificados com desempenho unitário.

Quanto às avaliações, elas se deram na faixa de 8k (8192) partículas até 1024k, confrontando os desempenhos relativos dos três métodos sugeridos (1 de CPU e 2 de GPU), em dois sistemas (descritos no anexo D). No método baseado em CPU, foram computados os tempos decorridos nas transferências de dados entre memória de vídeo e memória

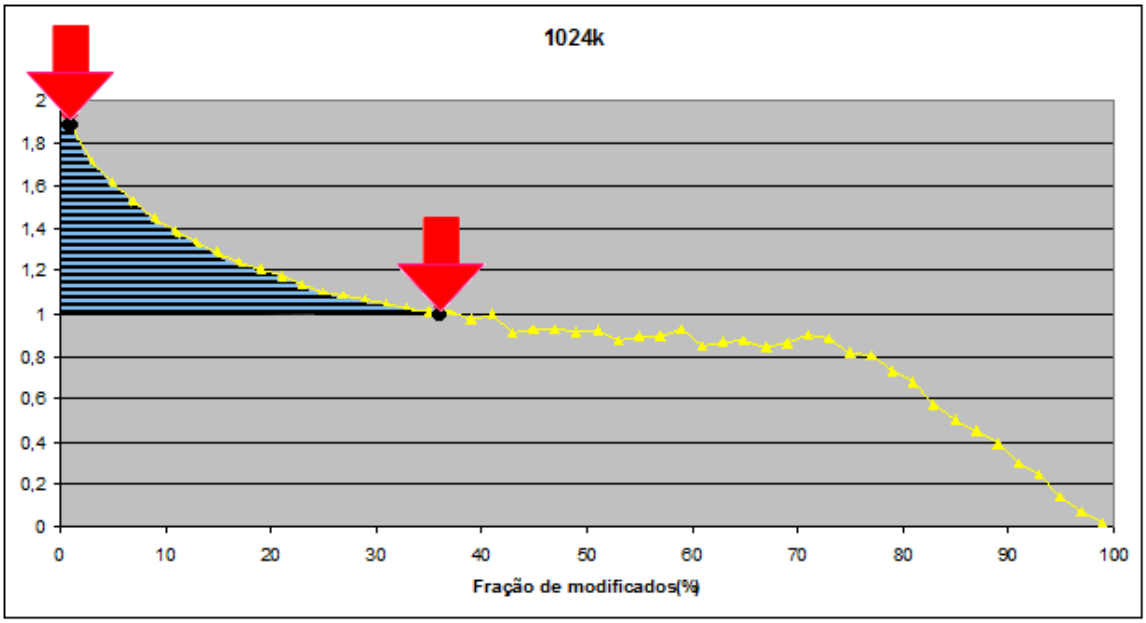

Figura 4.1: Exemplo de região de desempenho útil (área listrada).

principal. Os testes abrangeram também a diferenciação de comprimento de chaves (20, 24, 28 e 32 bits), pelo fato de o algoritmo *radix sort* tirar proveito dessa diferenciação, já que afeta o número de passos. Exemplificando, a adoção de uma chave de 20 bits, o que atende a uma simulação numa grade de $64 \times 64 \times 64$ células (18 bits), permite a ordenação do *array* pelo *radix* em 5 passos em lugar de 8 para uma chave de 32 bits.

Comportamento no Sistema 1

Os resultados mostram que, nas faixas iniciais (8k a 16k pares), apenas o método otimizado baseado em CPU consegue ser competitivo perante o método mais rápido de GPU (no caso, o *bitonic sort*), isso, para uma pequena região de desempenho útil, ou seja, aplicável a animações bem suaves. Também para animações desse tipo, com 32k pares a serem ordenados, o método baseado em CPU consegue se mostrar mais rápido apenas quando as chaves possuem mais de 20 bits , ratificando a explicação na seção 2.7.2. A partir de 32k pares, os dois métodos otimizados direcionados ao processamento paralelo começam a mostrar benefícios, evidenciados pelas crescentes regiões de desempenho útil. Nesse quadro, o método que emprega operações atômicas (GPU2) se mostra superior ao método GPU1 até 512k pares (onde os desempenhos se assemelham); e com 1024k pares, o método GPU1 se torna o mais eficiente.

Comportamento no Sistema 2

Nesse sistema, destaca-se que o método baseado em CPU, até 16k pares, não supera os de GPU em nenhuma circunstância. A partir de 32k pares, seus benefícios ficam restritos

a animações suaves. As sugestões GPU1 e GPU2 começam a ter regiões de desempenho útil a partir de 16k pares, chegando ao pico de desempenho relativo de 4.5 (chaves de 32 bits e 256k pares). Diferentemente do comportamento no Sistema 1, o método GPU2 consegue ser superior ao método GPU1 em quase todas as circunstâncias.

Comportamentos comuns aos Sistemas 1 e 2

Nos dois sistemas avaliados, observou-se que, de forma geral, o aumento do tamanho da chave (número de bits) resultou em aumentos dos picos de desempenho e das regiões de desempenho útil (em se comparando para um mesmo número de pares). Esse comportamento advém principalmente do aumento do número de passos empregados no *radix sort*, que o torna mais custoso temporalmente.

Sobre o método GPU1, observa-se que o decaimento de desempenho se dá na forma aproximada de escada com degraus cada vez mais largos. Isso ocorre porque os *sorters* empregados na ordenação da tabela de itens modificados são implementados para operar com número de itens potência de dois, ou seja, se o número de itens não for uma potência desse tipo, será promovido à potência de dois mais próxima.

4.2 Comparação qualitativa

O objetivo dessa comparação foi verificar o impacto nos desempenhos dos algoritmos de ordenação sugeridos (CPU 3.4, GPU1 3.5.1 e GPU2 3.5.2) e referências (*bitonic sort* e *radix sort*) em contextos práticos de simulações (inclusive com renderização), tendo em vista que outras tarefas poderão ser muito custosas computacionalmente como, por exemplo, a pesquisa de vizinhança na *narrow phase*. O *software* empregado nos testes seguiu a mesma linha do demo *particles* da NVIDIA para OpenCL [33], inclusive na adoção de mesmas dimensões para as partículas, ficando as diferenças, no protótipo desenvolvido, por conta da implementação de resposta mecânica um pouco mais complexa e renderização executada a cada 3 passos de integração (veja Fig. 4.2). As avaliações, que consistiram na medida de tempo na produção de 10000 quadros consecutivos, foram conduzidas no sistema D.1, num subespaço dividido numa grade de 2M células cúbicas (128 subdivisões por eixo). Resultados como taxa de exibição FPS (*frames per second*) e desempenho relativo foram calculados a partir dessa medição de tempo. O desempenho relativo foi expresso como a razão entre duas taxas FPS, onde a taxa que se assumiu como referência (divisor) foi a da animação que empregou o *radix sort* na mesma condição de cinemática (de mesmo *time step*). Os contextos de testes foram os seguintes:

1. 256k partículas (veja Fig. 4.3), aceleração apenas pelo contato com os limites da caixa, células de dimensões $6.188R \times 6.188R \times 6.188R$ (onde R denota o raio da

partícula) e dois passos de tempo (*time steps*) a saber, 0.0010 e 0.0005 , com resultados mostrados nas tabelas 4.1 e 4.2, respectivamente, e comparação de desempenhos na figura 4.6. As fatias de tempo, discriminadas por método e por *time step*, são mostradas nas figuras 4.10 e 4.11.

2. 512k partículas (veja Fig. 4.4), com acelerações gravitacional uniforme, de contato entre partículas e de contato com os limites da caixa; células de dimensões $2.063R \times 2.063R \times 2.063R$ e *time step* 0.0010 . Resultados na tabela 4.3 e comparação de desempenhos na figura 4.7. As fatias de tempo discriminadas por método são mostradas na figura 4.12.

3. 128k partículas (veja Fig. 4.5), com acelerações gravitacional em dois pontos, de contato entre partículas e de contato com os limites da caixa; células de dimensões $4.125R \times 4.125R \times 4.125R$ e *time step* 0.0005 . Resultados na tabela 4.4 e comparação de desempenhos na figura 4.8. As fatias de tempo discriminadas por método são mostradas na figura 4.13.

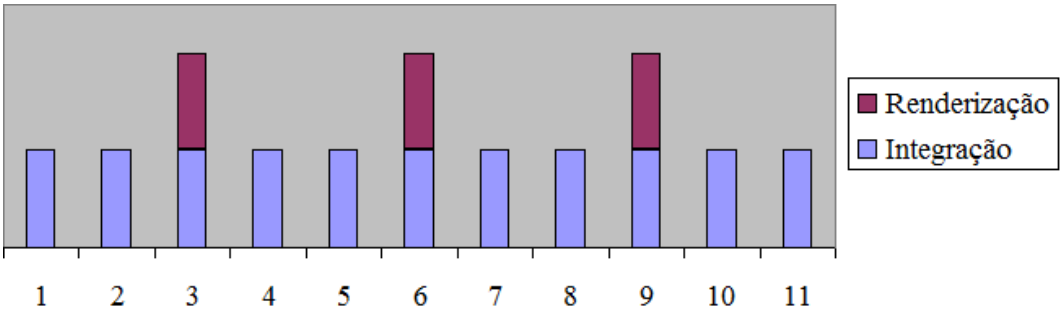

Figura 4.2: Esquema de sucessão de *frames* adotado nas animações.

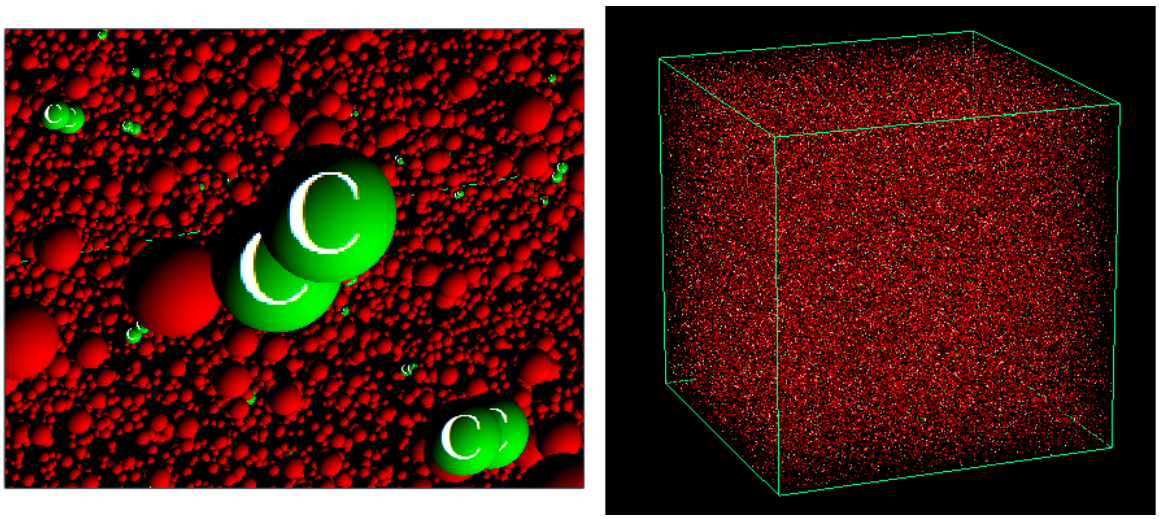

Figura 4.3: Colisão sinalizada pela mudança de cor (de vermelho para verde) e aplicação de textura (letra C) sobre as partículas envolvidas.

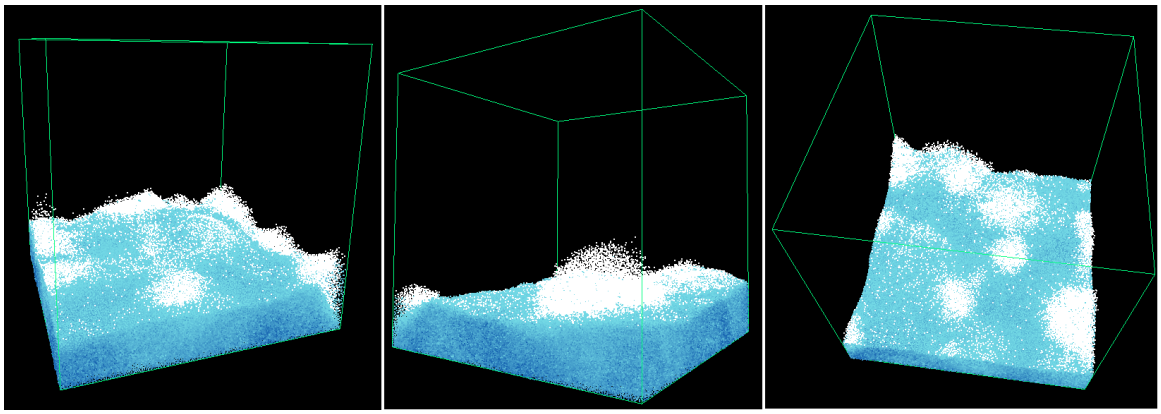

Figura 4.4: Simulação de “mar agitado”

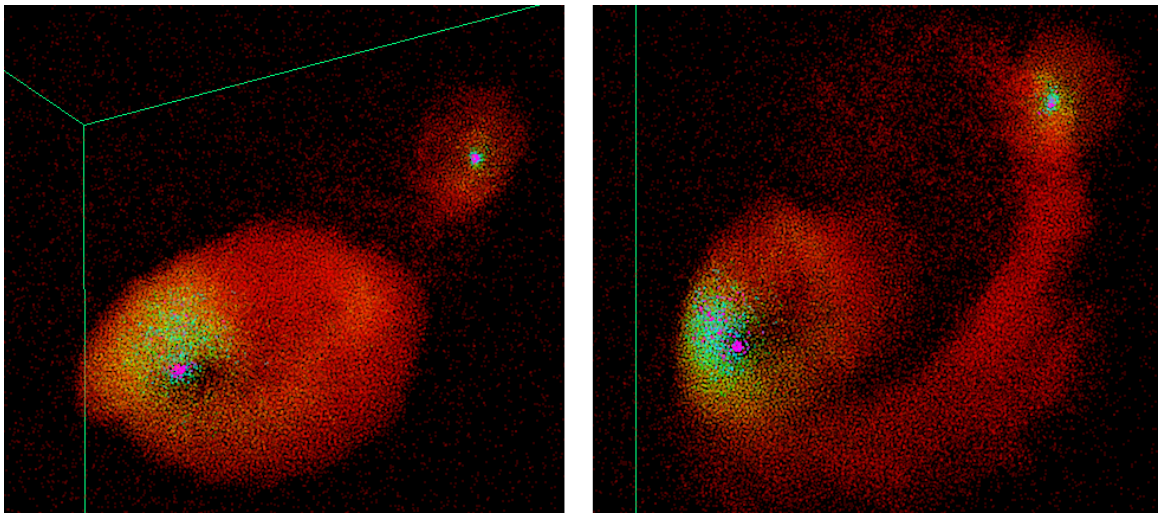

Figura 4.5: Simulação “efeito gravitacional”

Os resultados desses experimentos permitem as seguintes conclusões:

1. Nos testes do contexto 1, no qual a distribuição de partículas ao longo da animação permaneceu uniforme, os métodos empregados se comportaram conforme a previsão (veja figura B.6), servindo também para demonstrar que a diminuição do *time step* traz melhora de performance para os métodos otimizados;

2. Na simulação “mar agitado”, os métodos de ordenação otimizados (com exceção do baseado em CPU) deram uma margem de ganho de velocidade na ordem de 10%. O resultado foi bastante influenciado pela “aceleração da gravidade”, responsável pela sobrecarga de população em células inferiores, aumentando a tarefa na *narrow phase* (veja Fig. 4.12) perante simulações com distribuição uniforme de partículas no domínio;

3. Da simulação “efeito gravitacional”, apesar de os testes sobre os *sorters* indicarem ampla superioridade do método baseado em instruções atômicas (GPU2) para o mesmo contexto (veja figura B.6), o método GPU1 se saiu ligeiramente melhor. A

conclusão sobre esse resultado foi que o tipo de simulação “atração de partículas”, que tem por característica direcionar a migração para poucos pontos, ocasiona muitas inclusões sobre um mesmo código de *hash*, o que acaba serializando muitos acessos a um mesmo contador. Esse quadro acarreta também em mais itens a serem ordenados para uma mesma chave mantida (Fig. 3.12). Esses dois fatos permitem explicar a queda de performance.

Método	Tempo de simulação (s)	FPS (Hz)
CPU	115.43	86.63
bitonic sort	81.17	123.20
radix sort	62.21	160.75
GPU1	54.71	182.78
GPU2	54.04	185.05

Tabela 4.1: Simulação de 256k partículas ao longo de 10000 *frames* com *time step* igual a 0.010.

Método	Tempo de simulação (s)	FPS (Hz)
CPU	99.21	100.80
bitonic sort	80.99	123.47
radix sort	62.18	160.82
GPU1	51.80	193.05
GPU2	52.36	190.99

Tabela 4.2: Simulação de 256k partículas ao longo de 10000 *frames* com *time step* igual a 0.005.

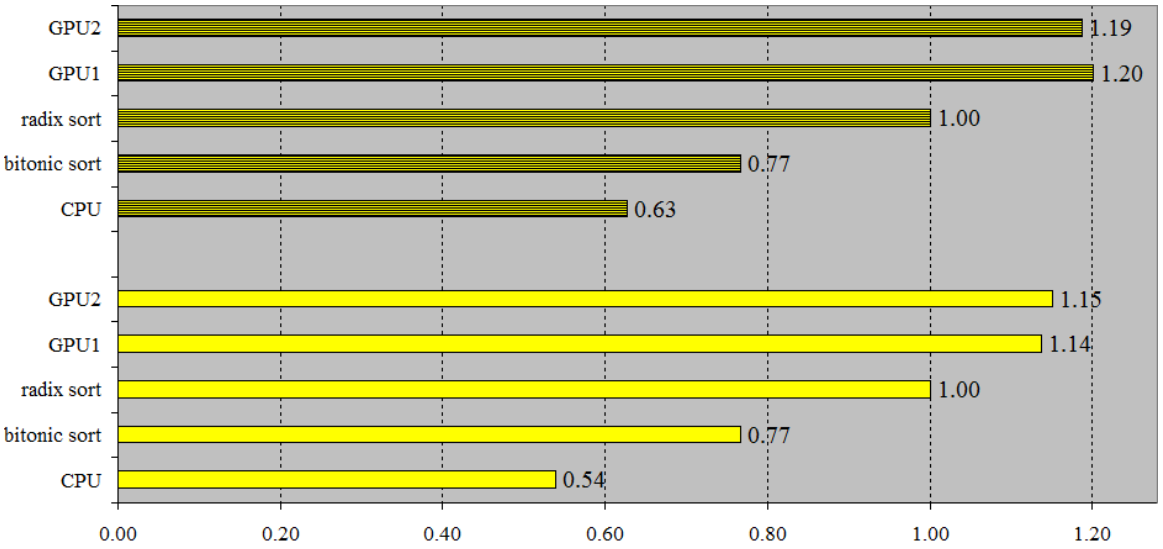

Figura 4.6: Desempenhos relativos obtidos nas avaliações do contexto 1, nos *time steps* 0.0010 (sem listras) e 0.0005 (com listras).

Método	Tempo de simulação (s)	FPS (Hz)
CPU	305.47	32.74
bitonic sort	201.64	49.59
radix sort	153.58	65.11
GPU1	138.33	72.29
GPU2	140.23	71.31

Tabela 4.3: Simulação de 512k partículas ao longo de 10000 *frames* com *time step* igual a 0.010.

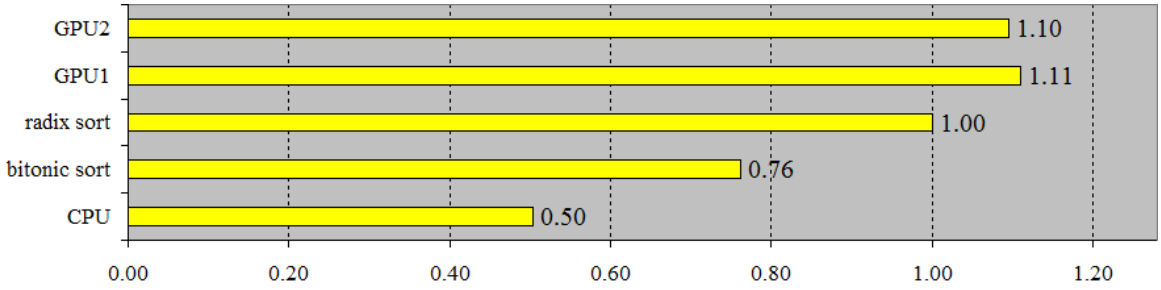

Figura 4.7: Desempenhos relativos obtidos nas avaliações do contexto 2.

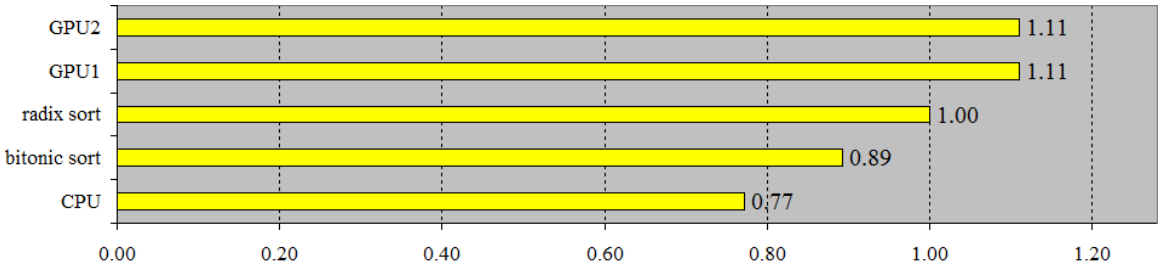

Figura 4.8: Desempenhos relativos obtidos nas avaliações do contexto 3.

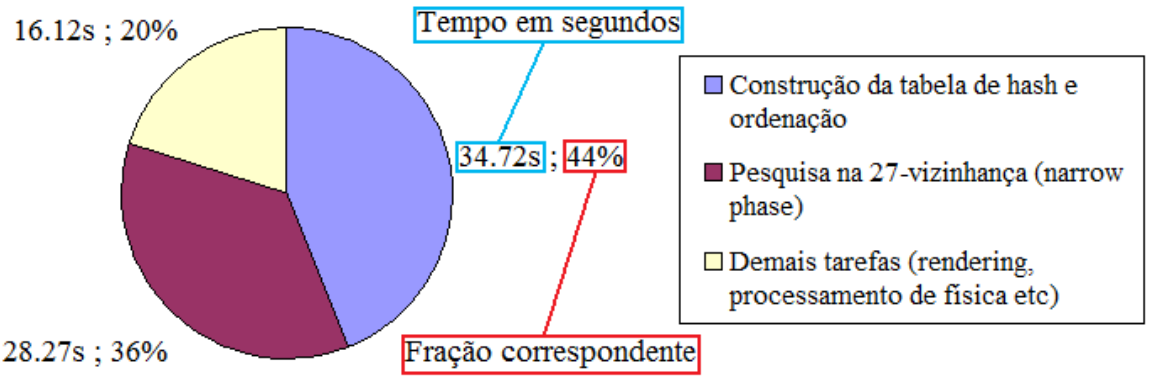

Figura 4.9: Simbologia adotada nos gráficos de fatias de tempo.

Método	Tempo de simulação (s)	FPS (Hz)
CPU	79.11	126.41
bitonic sort	68.35	146.31
radix sort	61.02	163.88
GPU1	54.94	182.02
GPU2	54.99	181.85

Tabela 4.4: Simulação de 128k partículas ao longo de 10000 *frames* com *time step* igual a 0.005.

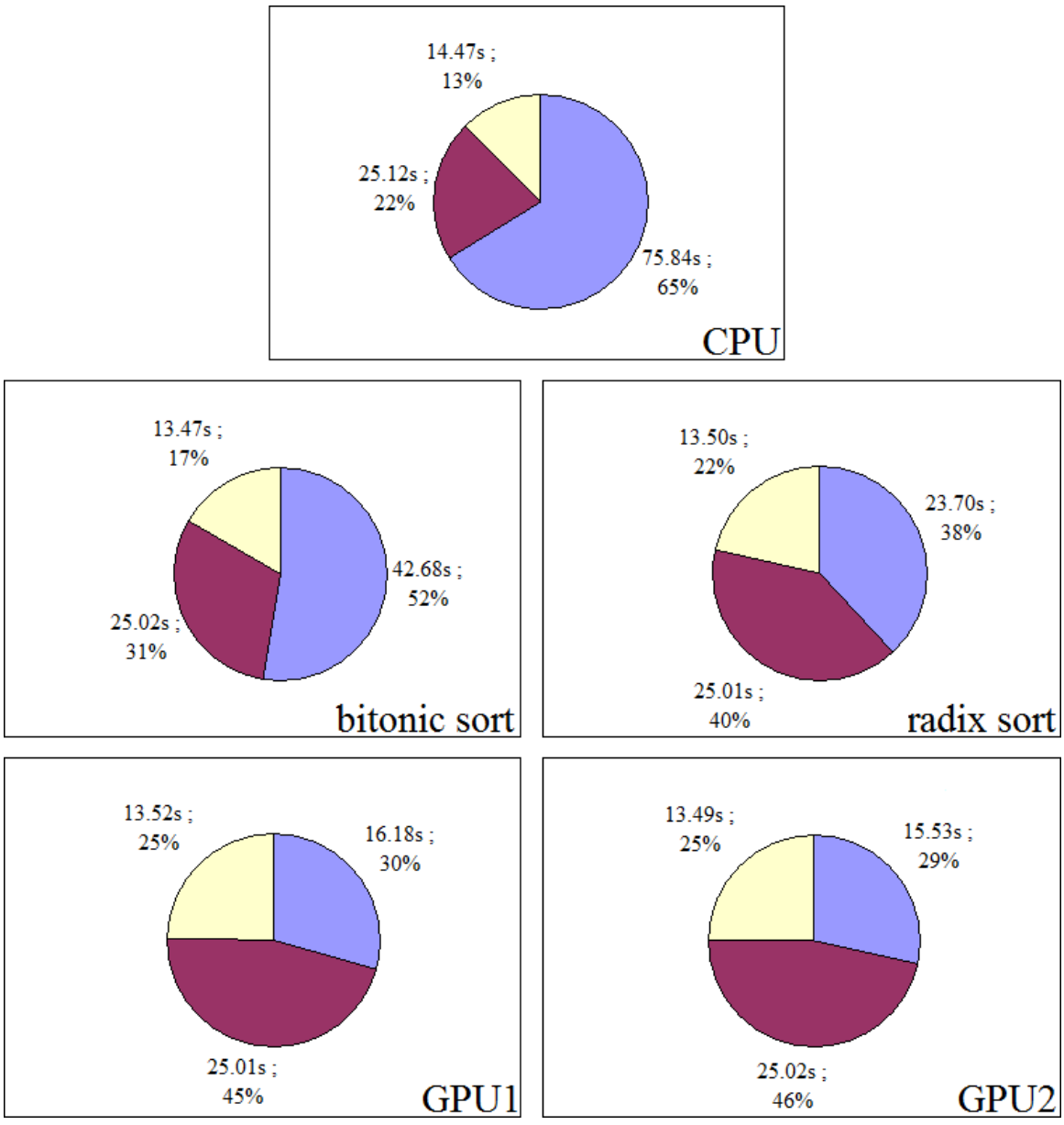

Figura 4.10: Fatias de tempo no contexto 1 (*time step* = .0010).

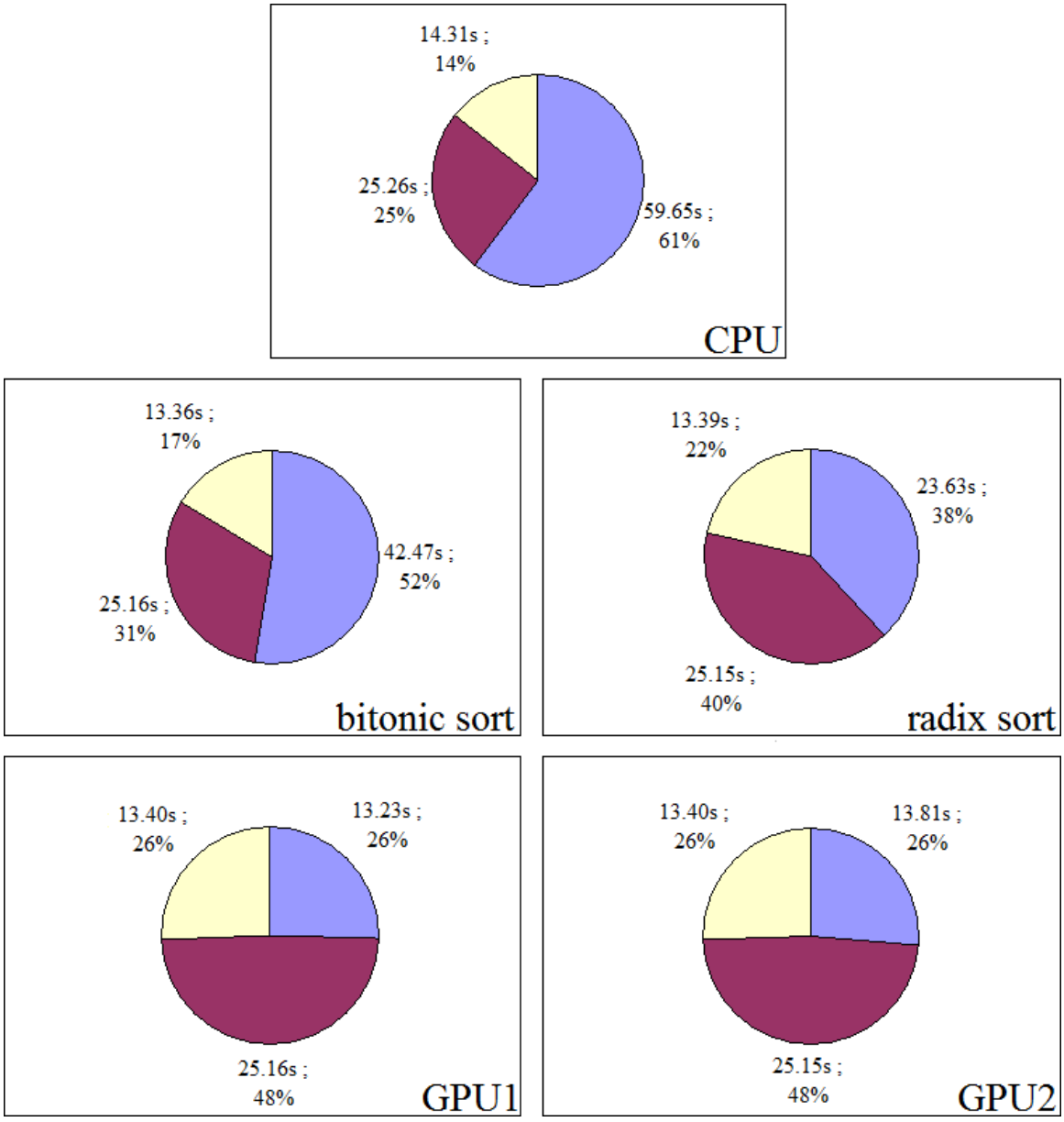

Figura 4.11: Fatias de tempo no contexto 1 (*time step* = .0005).

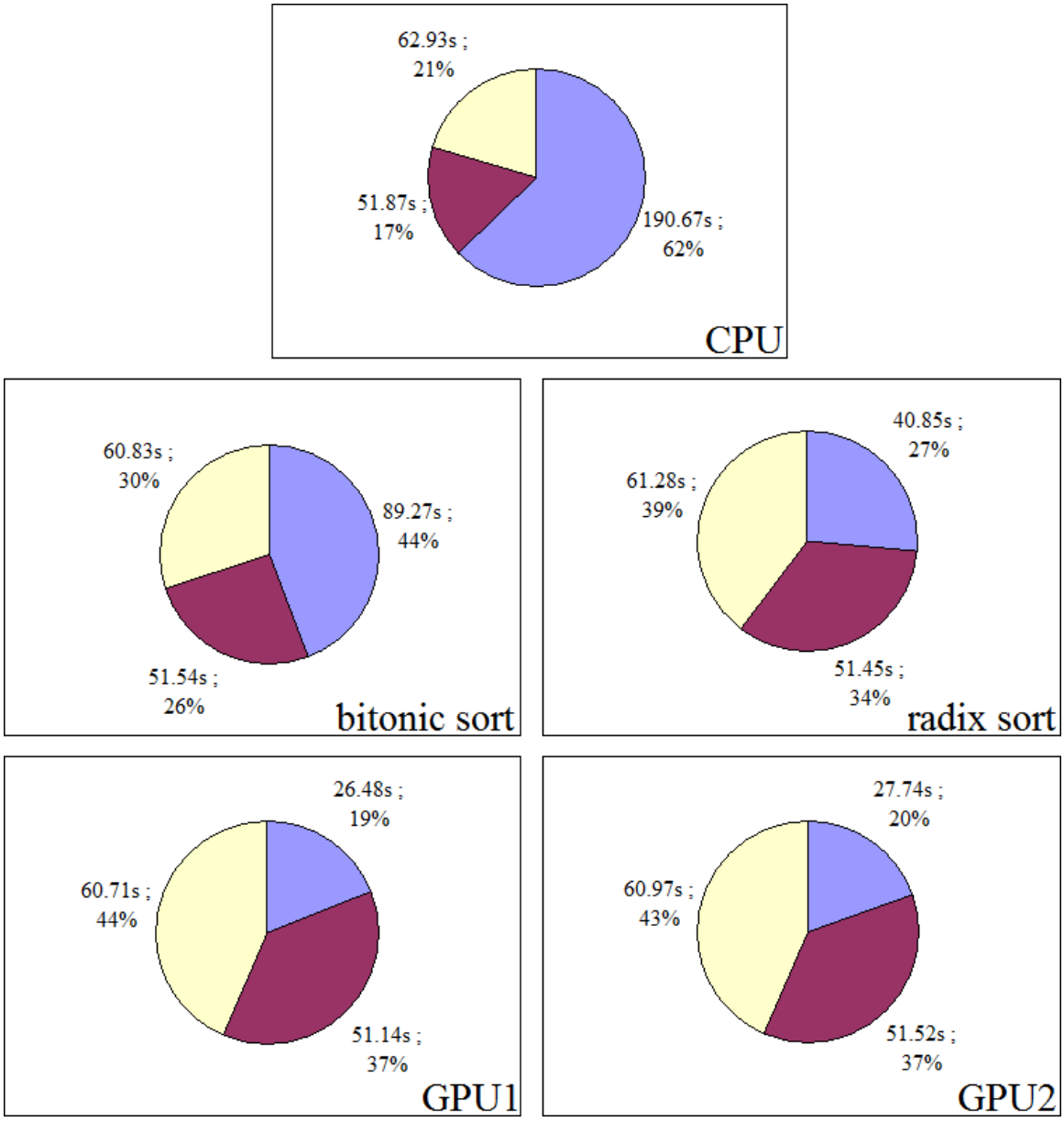

Figura 4.12: Fatias de tempos no contexto 2.

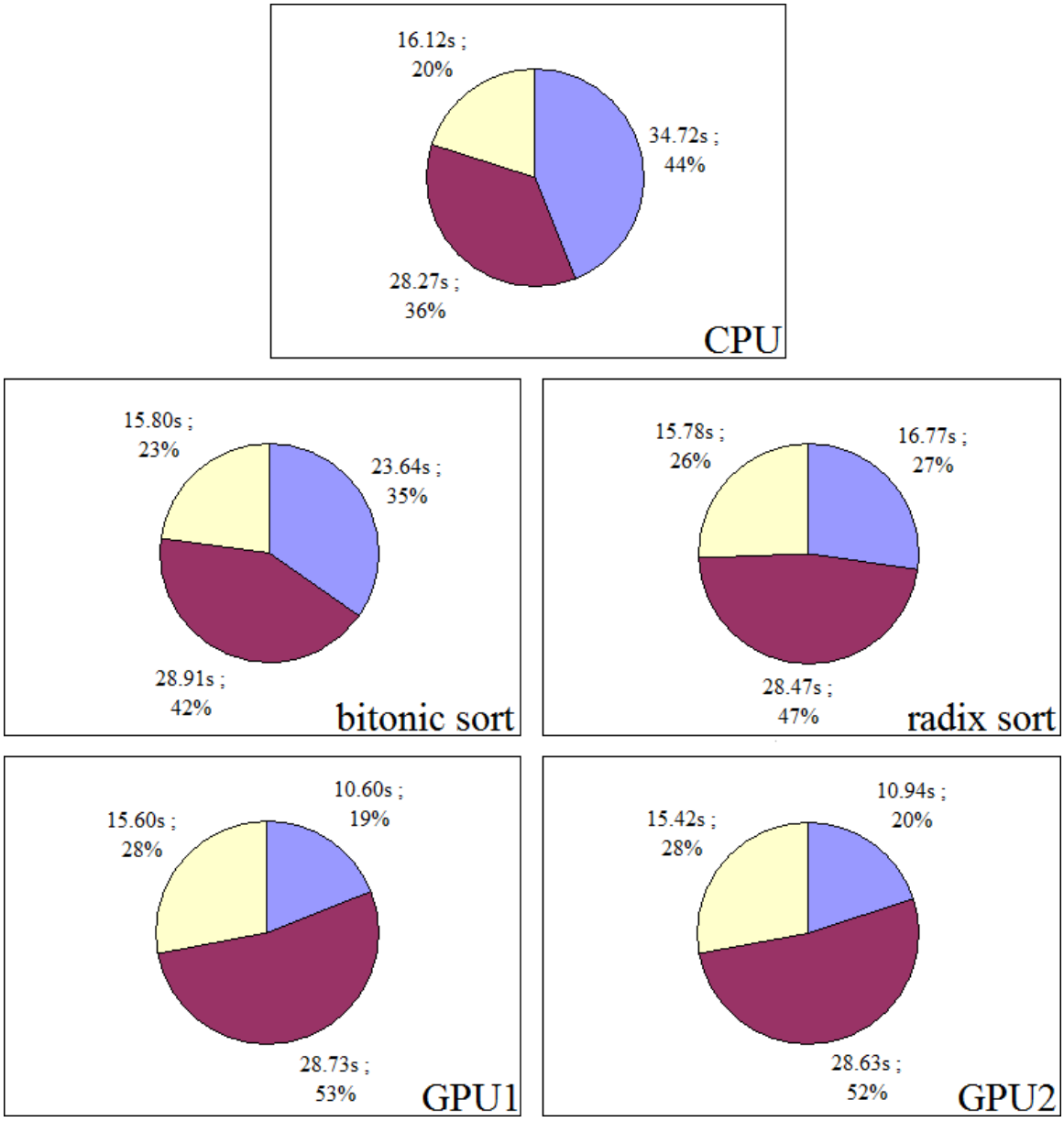

Figura 4.13: Fatias de tempo no contexto 3.

4.3 Decorrências do aproveitamento da continuidade de *frames* (quando em um espaço subdividido em grade).

As decorrências a seguir servem para ilustrar algumas relações causais numa simulação típica com impacto direto em desempenho. Aqui, as recíprocas não são sugeridas, mas podem ocorrer. Outras implicações não mencionadas poderão também existir, como, por exemplo, a adoção de maior *time step* para acelerar simulações, que poderá acarretar em erro na DC ou na estabilidade numérica do processo.

1. Transições suaves
    - transições suaves $\rightarrow$ menos mudança de células
    - menos mudança de células $\rightarrow$ muitos códigos de hash mantidos
    - muitos códigos de hash mantidos $\rightarrow$ tabela "bastante" ordenada
2. *Time step* elevado
    - *time step* elevado $\rightarrow$ menos continuidade
    - menos continuidade $\rightarrow$ mais mudança de células
    - mais mudança de células $\rightarrow$ muitos códigos de hash modificados
    - muitos códigos de hash modificados $\rightarrow$ tabela "pouco" ordenada
3. Células "grandes"
    - células "grandes" $\rightarrow$ tabela "bastante" mantida
    - células "grandes" $\rightarrow$ DC de *narrow phase* mais custosa, já que cada célula poderá comportar muitos itens

Capítulo 5

Conclusão

Os resultados apresentados no capítulo 4 demonstraram que o aproveitamento das coerências temporal e espacial são efetivos na aceleração de simulações baseadas em partículas, inclusive na versão puramente sequencial (CPU). O ponto negativo da proposta é a necessidade de emprego de *buffers* auxiliares. Isso, perante os pequenos ganhos de desempenho global em simulações que tenham processamento intenso na *narrow phase*, poderá ser o fator preponderante pela opção por soluções que não aproveitem as coerências. Uma constatação digna de menção é que não apenas o número de partículas será fator determinante na escolha do melhor método. Fatores como tamanho de chaves, sistema onde será executado e tipo de simulação são de grande impacto no desempenho final, sugerindo que a opção por algum método se faça por algum tipo de avaliação contextual, por exemplo, através de um *benchmark*.

Sobre a solução sequencial, observa-se ainda que ela poderá ser muito mais rápida com a diminuição (ou eliminação) do gargalo da transferência de dados entre memória da placa de vídeo e memória principal, o que talvez se dê em sistemas que compartilhem memória principal como RAM de vídeo. Para ilustrar tal possibilidade, no sistema D.2, na faixa de 16k pares com 20 bits de chave, foi realizada uma avaliação do método sequencial sem o *overhead* de transferência entre memórias, obtendo-se como resultado o *speedup* máximo na ordem de 6, o que pode tornar essa opção bastante atraente em determinadas simulações (gráfico 5.2 conforme simbologia da figura 5.1).

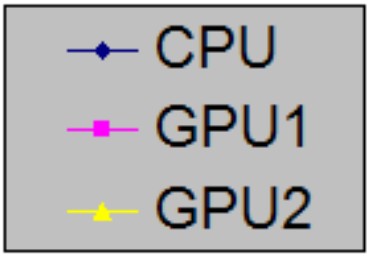

Figura 5.1: Correspondência nos gráficos: CPU (Sec. 3.4), GPU1 (Subsec. 3.5.1) e GPU2 (Subsec. 3.5.2)

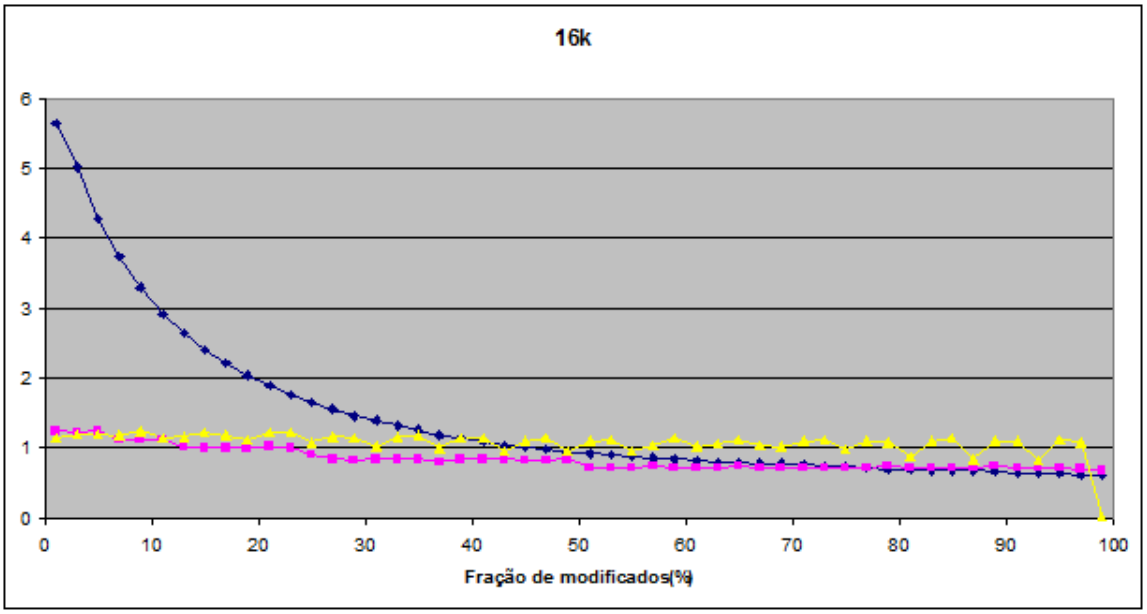

Figura 5.2: 16k pares (20 bits) ordenados **sem** *overhead* de transferência de memória (com *overhead*, no anexo C)

Capítulo 6

Trabalhos Futuros

O emprego da função de *hash* baseado em campo de bits (apresentado na seção 2.4), numa grade 3D, traz decorrências de incrementos fixos de elementos na tabela de *hash*, o que acarreta em 27 possibilidades de incremento (supondo deslocamentos das partículas para, no máximo, células adjacentes na sucessão de *frames*). Isso resultaria em até 27 tabelas de pares ordenadas a sofrerem *merge*, podendo trazer mais eficiência ao processo. Um aspecto interessante do uso dos métodos de ordenação é que eles possuem diversas aplicações, além de auxílio à detecção de colisão: corretude do efeito de *blend* (transparência), simplificação de tabelas (retirada de dados duplicados), entre outros; tornando a contribuição deste trabalho estendível a outras áreas que usam ordenação de tabelas suscetíveis a atualizações. Outra extensão possível deste trabalho é o emprego de outros métodos para a "soma de prefixos", *split*, *merge*, outros *sorters* etc., para comparação nos mesmos contextos e em outras faixas de operação (número de pares a serem ordenados maiores que 1M). Afastando-se dos métodos de ordenação, mas ainda com foco nas simulações baseadas em partículas, outro alvo de otimização poderia ser o processamento de *narrow phase*, especificamente, na pesquisa da vizinhança. Como mostrado nos gráficos de fatias de tempo (figuras 4.10, 4.11, 4.12 e 4.13), a parcela destinada a essa pesquisa é um dos principais gargalos desse tipo de animação.

Referências Bibliográficas

[1] HOUSER, T. "Overview of 3D Object Representation - COS 426". Princeton Univercity, 2006. Disponível em: <http://www.cs.princeton.edu/academics/coursesched/spring06>.

[2] BRADSHAW, G. "Sphere-Tree Construction Toolkit". feb 2003. Disponível em: <http://isg.cs.tcd.ie/spheretree/#pubs>.

[3] NGUYEN, H. *Gpu Gems 3*. Addison-Wesley Professional, 2007. ISBN: 9780321545428. Disponível em: <http://http.developer.nvidia.com/GPUGems3/gpugems3_pref01.html>.

[4] SUTTER, H. "The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software", *Dr. Dobb's Journal*, v. 30, n. 3, 2005.

[5] GAVAN, V. "How to Create Sequence Photos". http://www.photopoly.net/how-to-create-sequence-photos/, 2012.

[6] WIKIPEDIA. "Simulação — Wikipedia, The Free Encyclopedia". http://pt.wikipedia.org/wiki/Simula%C3%A7%C3%A3o, 2012.

[7] CLAVET, S., BEAUDOIN, P., POULIN, P. "Particle-based viscoelastic fluid simulation". In: *Proceedings of the 2005 ACM SIGGRAPH/Eurographics symposium on Computer animation*, SCA '05, pp. 219–228, New York, NY, USA, 2005. ACM. ISBN: 1-59593-198-8. doi: 10.1145/1073368.1073400. Disponível em: <http://doi.acm.org/10.1145/1073368.1073400>.

[8] IHMSEN, M., AKINCI, N., BECKER, M., etal. "A Parallel SPH Implementation on Multi-Core CPUs", *Comput. Graph. Forum*, v. 30, n. 1, pp. 99–112, 2011.

[9] DAVID KNOTT, M. *CInDeR Collision and Interference detection in real time using graphics hardware*. Tese de Mestrado, UBC, 2003.

[10] BACIU, G., WONG, W. S. "Image-Based Techniques in a Hybrid Collision Detector", *IEEE Transactions on Visualization and Computer Graphics*, v. 9, pp. 254–271, 2003. ISSN: 1077-2626. doi: http://doi.ieeecomputersociety.org/10.1109/TVCG.2003.10012.

[11] GOVINDARAJU, N. K., LIN, M. C., MANOCHA, D. “Fast and reliable collision culling using graphics hardware”. In: *In Virtual Reality Software and Technology (VRST*, pp. 2–9, 2004.

[12] COHEN, J. D., LIN, M. C., MANOCHA, D., etal. “I-COLLIDE: An interactive and exact collision detection system for large-scale environments”. In: *In Proc. of ACM Interactive 3D Graphics Conference*, pp. 189–196, 1995.

[13] LIN, M. C., GOTTSCHALK, S. “Collision Detection Between Geometric Models: A Survey”. In: *In Proc. of IMA Conference on Mathematics of Surfaces*, pp. 37–56, 1998.

[14] LENAERTS, T., ADAMS, B., DUTRÉ, P. “Porous flow in particle-based fluid simulations”. In: *ACM SIGGRAPH 2008 papers*, SIGGRAPH ’08, pp. 49:1–49:8, New York, NY, USA, 2008. ACM. ISBN: 978-1-4503-0112-1. doi: 10.1145/1399504.1360648. Disponível em: <http://doi.acm.org/10.1145/1399504.1360648>.

[15] AKENINE-MÖLLER, T., HAINES, E., HOFFMAN, N. *Real-Time Rendering 3rd Edition*. Natick, MA, USA, A. K. Peters, Ltd., 2008. ISBN: 987-1-56881-424-7.

[16] VAN DEN BERGEN, G. *Collision Detection in Interactive 3D Environments (The Morgan Kaufmann Series in Interactive 3D Technology)*. Morgan Kaufmann, out. 2003. ISBN: 155860801X. Disponível em: <http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20&path=ASIN/155860801X>.

[17] ERICSON, C. *Real-Time Collision Detection (The Morgan Kaufmann Series in Interactive 3-D Technology)*. Morgan Kaufmann, jan. 2005. ISBN: 1558607323. Disponível em: <http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20&path=ASIN/1558607323>.

[18] LUQUE, R. G. *Árvores BSP Semi-Ajustáveis*. Mestrado, Programa de Pós-Graduação em Computação, UFRGS, 2005. Disponível em: <http://www.lume.ufrgs.br/bitstream/handle/10183/17349/000715187.pdf?sequence=1>.

[19] DOBASHI, Y., MATSUDA, Y., YAMAMOTO, T., etal. “A Fast Simulation Method Using Overlapping Grids for Interactions between Smoke and Rigid Objects”. 2008.

[20] REDON, S. “Continuous Collision Detection for Rigid and Articulated Bodies”. In: *SIGGRAPH*, 2004. Disponível em: <http://i3d.inrialpes.fr/people/redon/papers/sigCourse2004.pdf>.

[21] WIKIPEDIA. “Collision detection — Wikipedia, The Free Encyclopedia”. http://en.wikipedia.org/wiki/Collision_detection, 2012.

[22] EDITORIAL, L. *LAROUSSE DICIONÁRIO DA LÍNGUA PORTUGUESA*. ÁTICA. ISBN: 9788508080779. Disponível em: <http://books.google.com.br/books?id=Mu1yQgAACAAJ>.

[23] GREEN, S. “Particle Simulation using CUDA”. NVIDIA - presentation packaged with CUDA Toolkit, 2010.

[24] SYLVAN, S. *Particle System Simulation and Rendering on the Xbox 360 GPU*. Tese de Mestrado, CHALMERS UNIVERSITY OF TECHNOLOGY, Department of Computer Science and Engineering, Division of Computer Engineering Göteborg, Sweden, 2007.

[25] EADLINE, D. *High Performance Computing For Dummies, Sun and AMD Special Edition*. 111 River Street Hoboken, NJ, Wiley Publishing, Inc., 2009.

[26] WRINN, M. “Is the free lunch really over?” *Intel Paper*, 2007. Disponível em: <http://software.intel.com/sites/default/files/m/d/4/1/d/8/Wrinn_Free_Lunch_part_1_Scalability.pdf>.

[27] CROW, T. S., FREDERICK, D., HARRIS, C. “Evolution of the Graphical Processing Unit”. 2004.

[28] WRIGHT, R. S., HAEMEL, N., SELLERS, G., etal. *OpenGL SuperBible: Comprehensive Tutorial and Reference*. Addison-Wesley Professional, 2010. ISBN: 0321712617, 9780321712615.

[29] WIKIPEDIA. “Graphics processing unit — Wikipedia, The Free Encyclopedia”. http://en.wikipedia.org/wiki/Graphics_processing_unit, 2012.

[30] CONRAD, D. F. “Análise da Hierarquia de Memórias em GPGPU - Bachelor’s thesis, UFRGS”. jul. 2010.

[31] PILKINGTON, N. C. V. “An Investigation into General Purpose Computation on Graphics Processing Units (GPGPU) - Bachelor’s thesis, Rhodes University”. maio 2007.

[32] COUMANS, E. “Porting Bullet to OpenCL - GDC 2010”. Sony Computer Entertainment US R&D, mar 2010.

[33] NVIDIA. “NVIDIA OpenCL SDK Code Samples”. http://developer.download.nvidia.com/compute/cuda/3_0/sdk/website/OpenCL/website/samples.html, 2012.

[34] BATCHER, K. E. “Sorting networks and their applications”. In: *Proceedings of the April 30–May 2, 1968, spring joint computer conference*, AFIPS ’68 (Spring), pp. 307–314, New York, NY, USA, 1968. ACM. doi: 10.1145/1468075.1468121. Disponível em: <http://doi.acm.org/10.1145/1468075.1468121>.

[35] LEISCHNER, N. “GPU algorithms for comparison-based sorting and for merging based on multiway selection”, 2010.

[36] MUELLER, R., TEUBNER, J., ALONSO, G. “Sorting networks on FPGAs”, *The VLDB Journal*, v. 21, n. 1, pp. 1–23, fev. 2012. ISSN: 1066-8888. doi: 10.1007/s00778-011-0232-z. Disponível em: <http://dx.doi.org/10.1007/s00778-011-0232-z>.

[37] LANG, H.-W. “Sorting networks — Bitonic sort”. http://www.iti.fh-flensburg.de/lang/algorithmen/sortieren/bitonic/bitonicen.htm, 2012.

[38] SARIPELLA, R. “Radix and Bucket Sort - CS566”. Disponível em: <http://www.docstoc.com/docs/83506308/Radix-and-Bucket-Sort-(PowerPoint)>.

[39] GRAEFE, G. “Implementing sorting in database systems”, *ACM Comput. Surv.*, v. 38, n. 3, set. 2006. ISSN: 0360-0300. doi: 10.1145/1132960.1132964. Disponível em: <http://doi.acm.org/10.1145/1132960.1132964>.

[40] AL-DARWISH, N. “Formulation and analysis of in-place MSD radix sort algorithms.” *J. Information Science*, v. 31, n. 6, pp. 467–481, 2005. Disponível em: <http://dblp.uni-trier.de/db/journals/jis/jis31.html#Al-Darwish05>.

[41] SATISH, N., HARRIS, M., GARLAND, M. “Designing efficient sorting algorithms for manycore GPUs”. In: *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, IPDPS ’09, pp. 1–10, Washington, DC, USA, 2009. IEEE Computer Society. ISBN: 978-1-

4244-3751-1. doi: 10.1109/IPDPS.2009.5161005. Disponível em: <http://dx.doi.org/10.1109/IPDPS.2009.5161005>.

[42] BLELLOCH, G. E. *Prefix Sums and Their Applications*. Relatório técnico, Synthesis of Parallel Algorithms, 1990.

[43] ESTIVILL-CASTRO, V., WOOD, D. "A survey of adaptive sorting algorithms", *ACM Comput. Surv.*, v. 24, n. 4, pp. 441–476, dez. 1992. ISSN: 0360-0300. doi: 10.1145/146370.146381. Disponível em: <http://doi.acm.org/10.1145/146370.146381>.

[44] GRÖLLER, E., PURGATHOFER, W. *Coherence in Computer Graphics*. Relatório técnico, 1992.

[45] HENDRIK, G. Z., LENSCH, H. P. A., MAGNOR, M., etal. "Multi-Video Compression in Texture Space using 4D SPIHT". In: *Proc. IEEE International Conference on Image Processing (ICIP04*, p. accepted., 2004.

[46] KUMAR, S., AZARTASH, H., BISWAS, M., etal. "Real-Time Affine Global Motion Estimation Using Phase Correlation and its Application for Digital Image Stabilization." *IEEE Transactions on Image Processing*, v. 20, n. 12, pp. 3406–3418, 2011. Disponível em: <http://dblp.uni-trier.de/db/journals/tip/tip20.html#KumarABN11>.

[47] MARTIN, I., PUEYO, X., TOST, D. "Frame-to-Frame Coherent Animation with Two-Pass Radiosity", *IEEE Transactions on Visualization and Computer Graphics*, v. 9, pp. 70–84, 1999.

[48] KALE, V., SOLOMONIK, E. "Parallel sorting pattern". In: *Proceedings of the 2010 Workshop on Parallel Programming Patterns*, ParaPLoP '10, pp. 10:1–10:12, New York, NY, USA, 2010. ACM. ISBN: 978-1-4503-0127-5. doi: 10.1145/1953611.1953621. Disponível em: <http://doi.acm.org/10.1145/1953611.1953621>.

[49] KIDER, J. *GPU as a Parallel Machine: Sorting on the GPU*. Lecture, University of Pennsylvania, mar 2005. Disponível em: <http://www.cis.upenn.edu/~suvenkat/700/lectures/19/sorting-kider.pdf>.

[50] OTTMANN, T. "Parallel Merging". N. 15, Lecture Notes in Computer Science - Advanced Algorithms & Data Structures, 2006. Disponível em: <http://electures.informatik.uni-freiburg.de/portal/download/3/6950/thm15%20-%20parallel%20merging.pdf>.

[51] HOVEMEYER, D., PUGH, W., SPACCO, J. “Atomic instructions in java”. In: *In Magnusson [14]*, pp. 133–154, 2002.

Apêndice A

Desempenhos do *bitonic sort* e do *radix sort* em relação a tabelas parcialmente ordenadas

O objetivo destas medições foi verificar a existência de tendência de desempenho associado ao ordenamento parcial, o que não foi identificada.

A.1 *Bitonic sort*

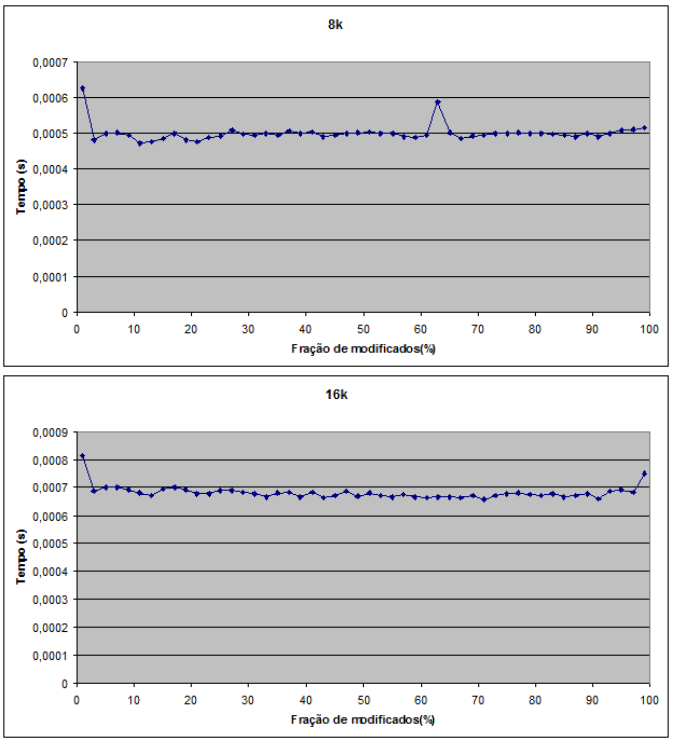

Figura A.1: Bitonic sort, 8k e 16k pares.

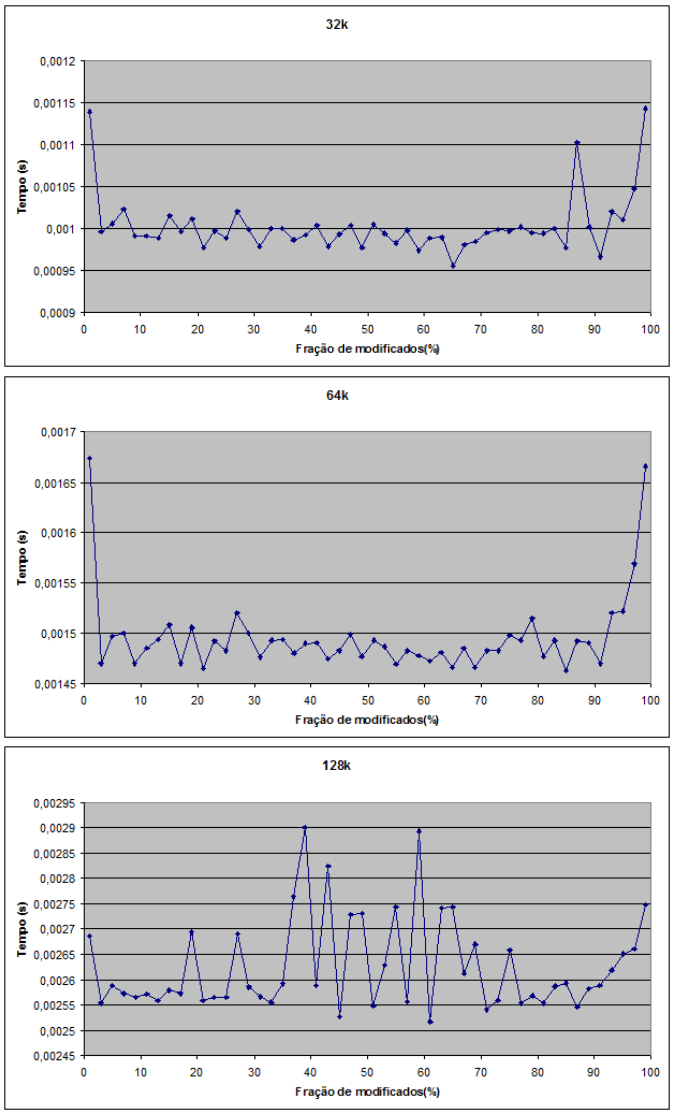

Figura A.2: Bitonic sort, 32k a 128k pares.

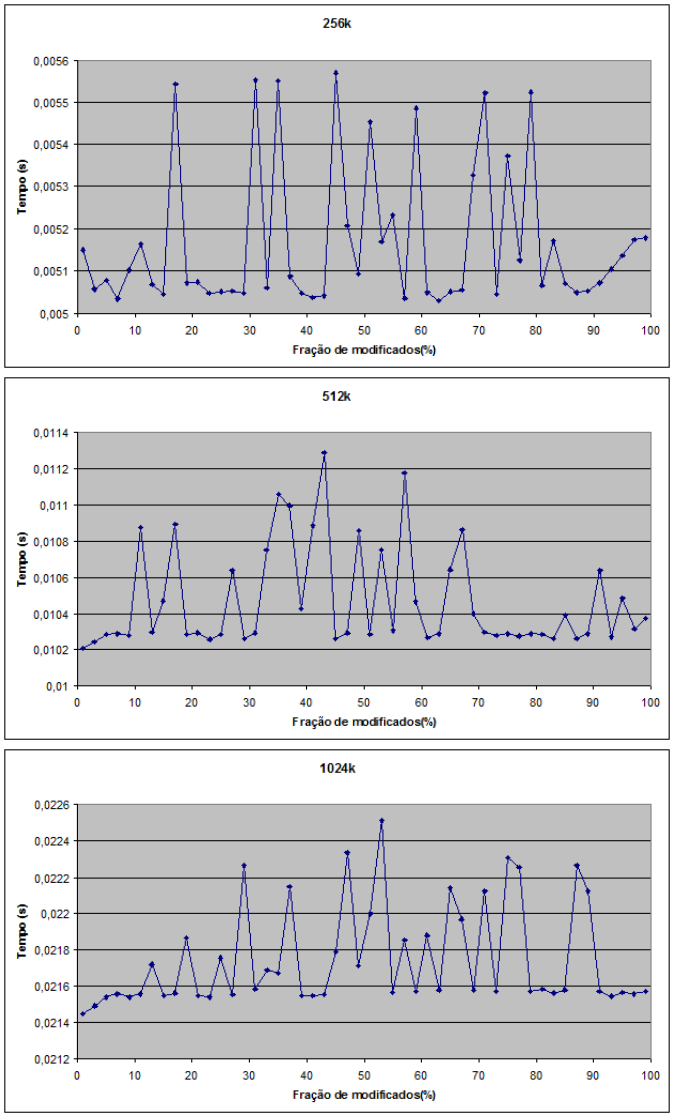

Figura A.3: Bitonic sort, 256k a 1024k pares.

A.2 *Radix sort*

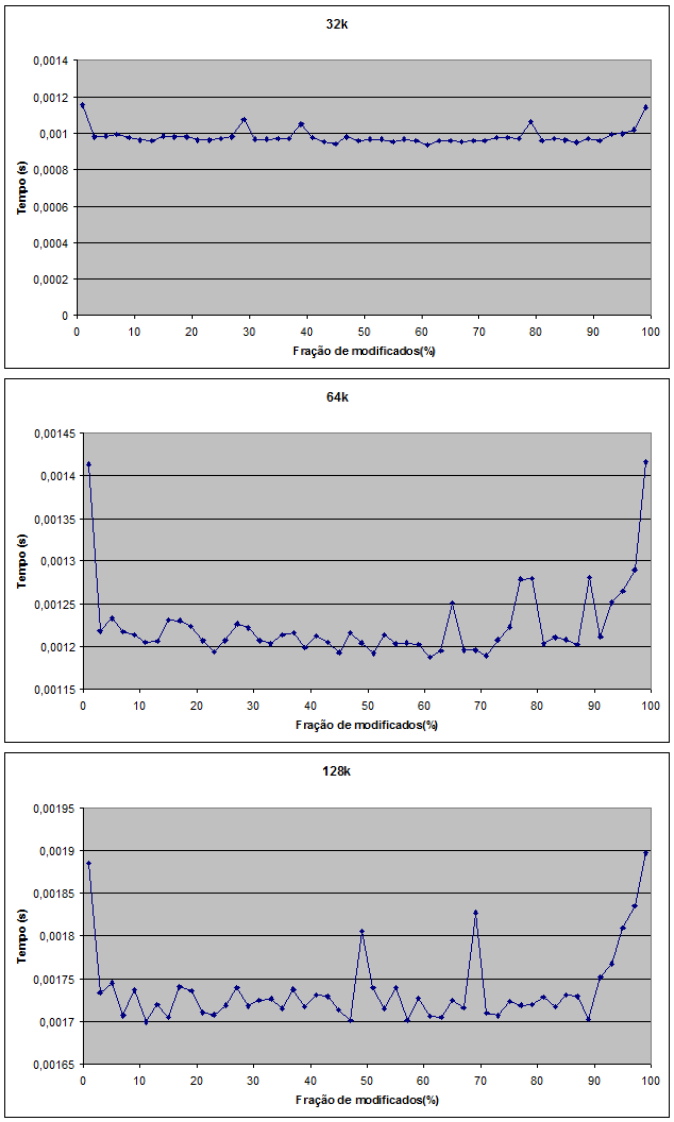

Figura A.4: Radix sort (chave de 20 bits), 32k a 128k pares.

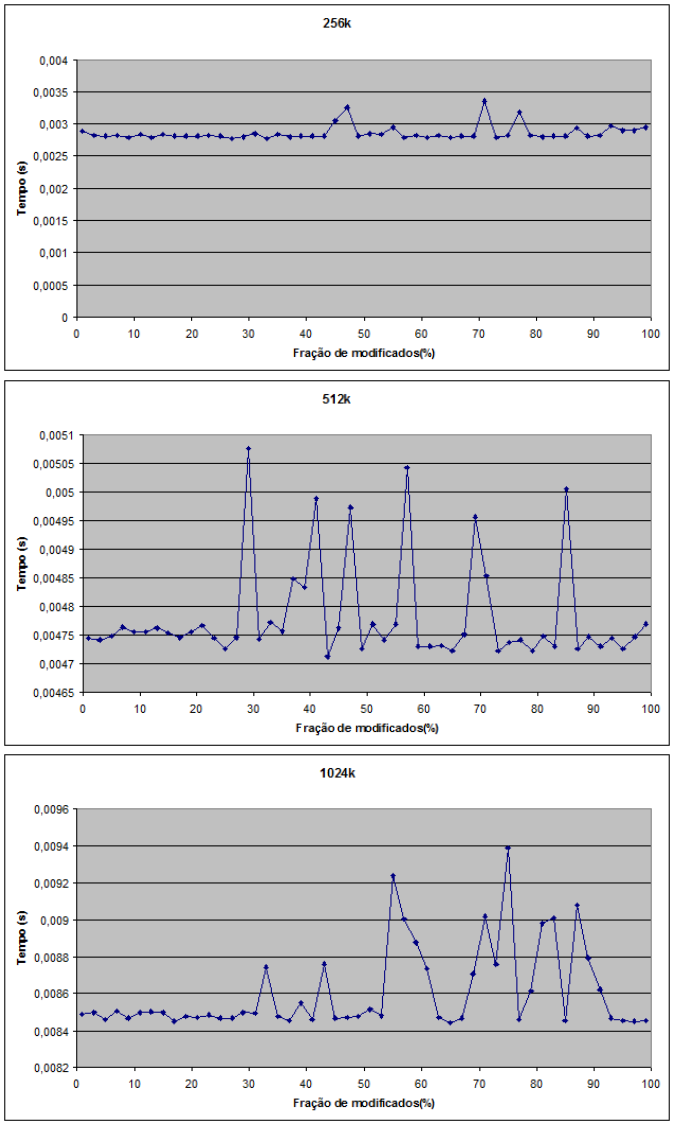

Figura A.5: Radix sort (chave de 20 bits), 256k a 1024k pares.

Apêndice B

Avaliação dos métodos no sistema D.1 com chaves de 20, 24, 28 e 32 bits

As comparações a seguir foram realizadas segundo a explicação da seção 4.1, adotando a simbologia da figura B.1.

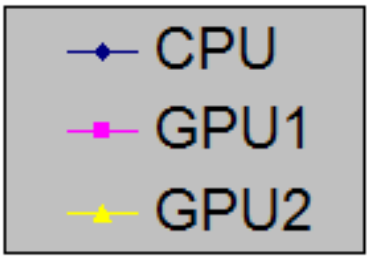

Figura B.1: Correspondência nos gráficos: CPU (Sec. 3.4), GPU1 (Subsec. 3.5.1) e GPU2 (Subsec. 3.5.2)

B.1 Chaves de 20 bits

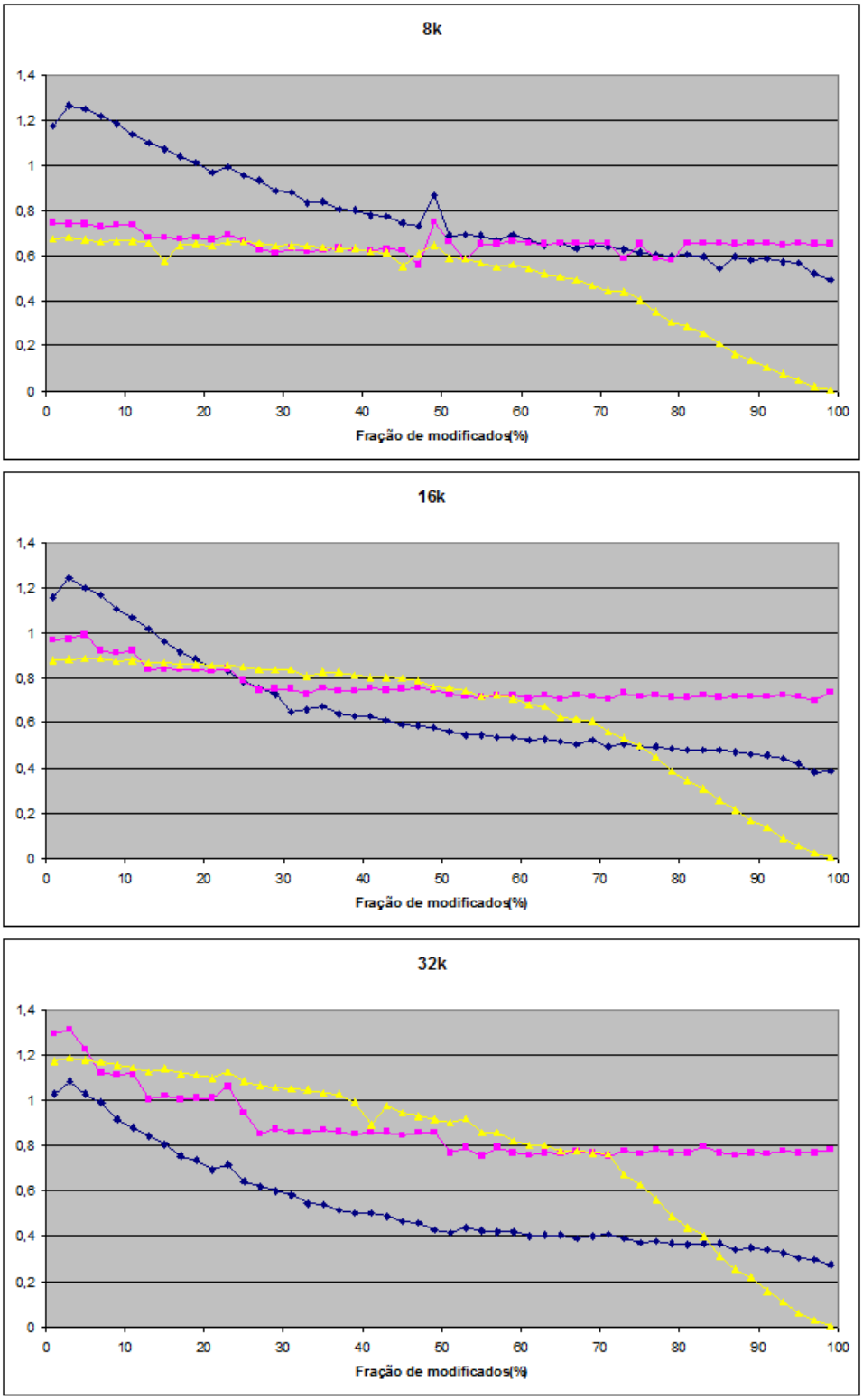

Figura B.2: Desempenhos relativos, 8k a 32k pares e chaves de 20 bits.

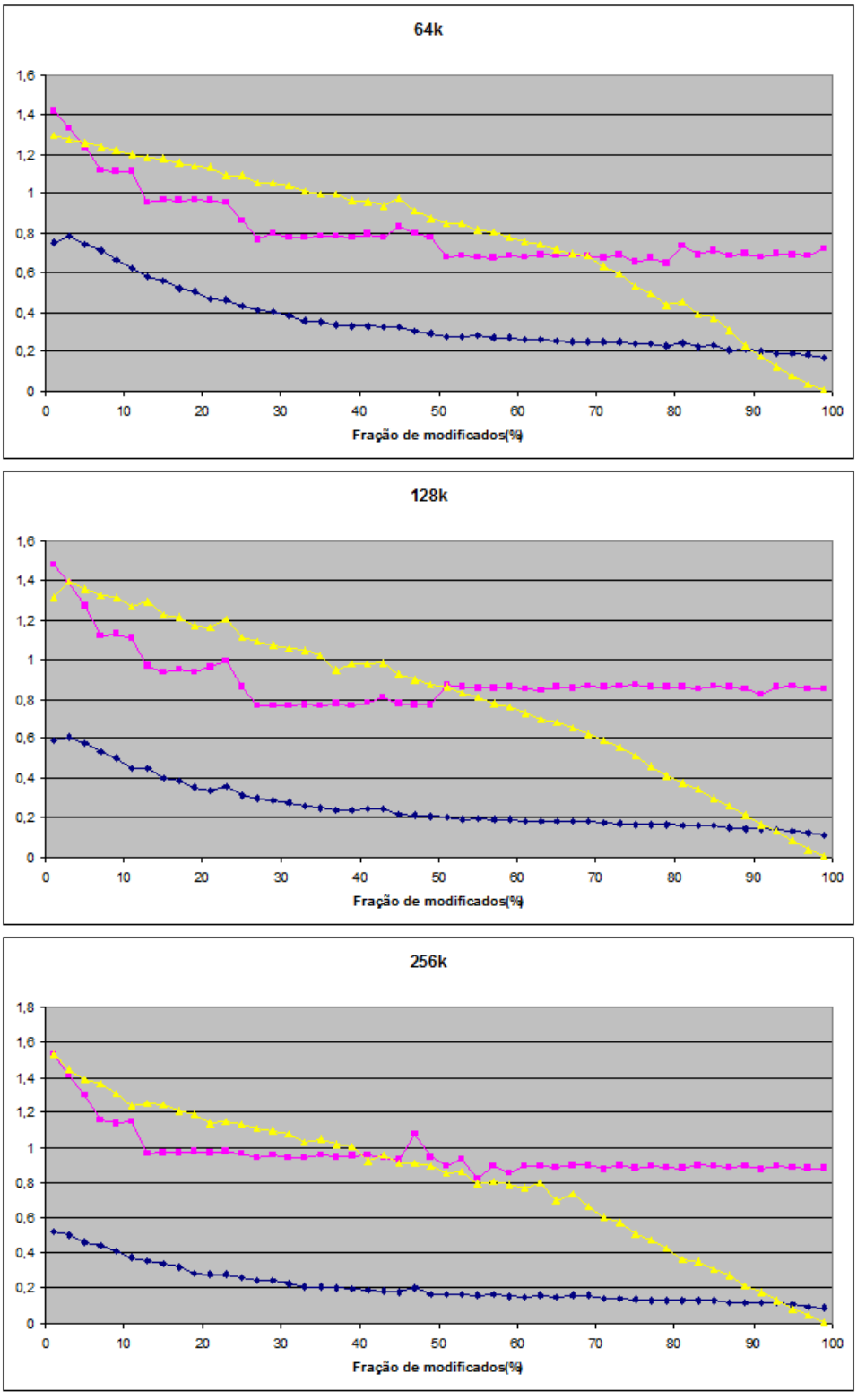

Figura B.3: Desempenhos relativos, 64k a 256k pares e chaves de 20 bits.

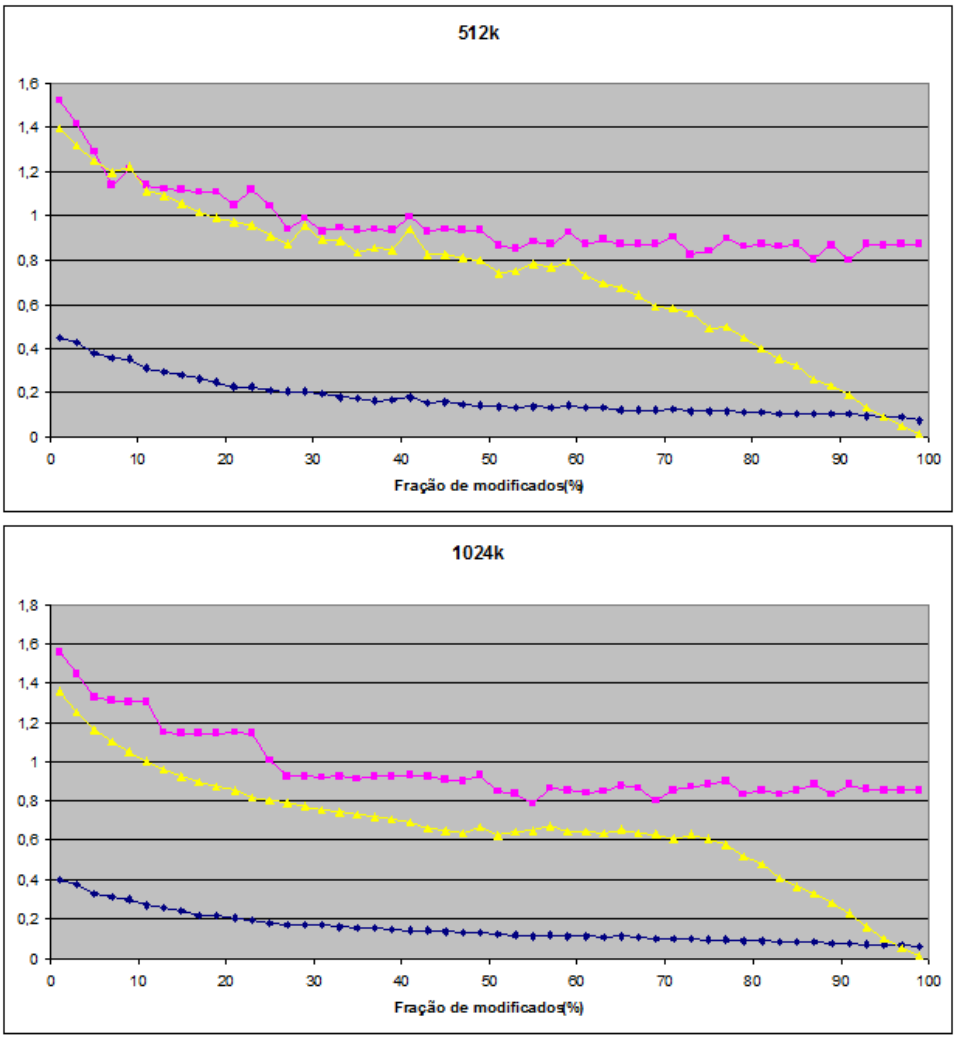

Figura B.4: Desempenhos relativos, 512k e 1024k pares e chaves de 20 bits.

B.2 Chaves de 24 bits

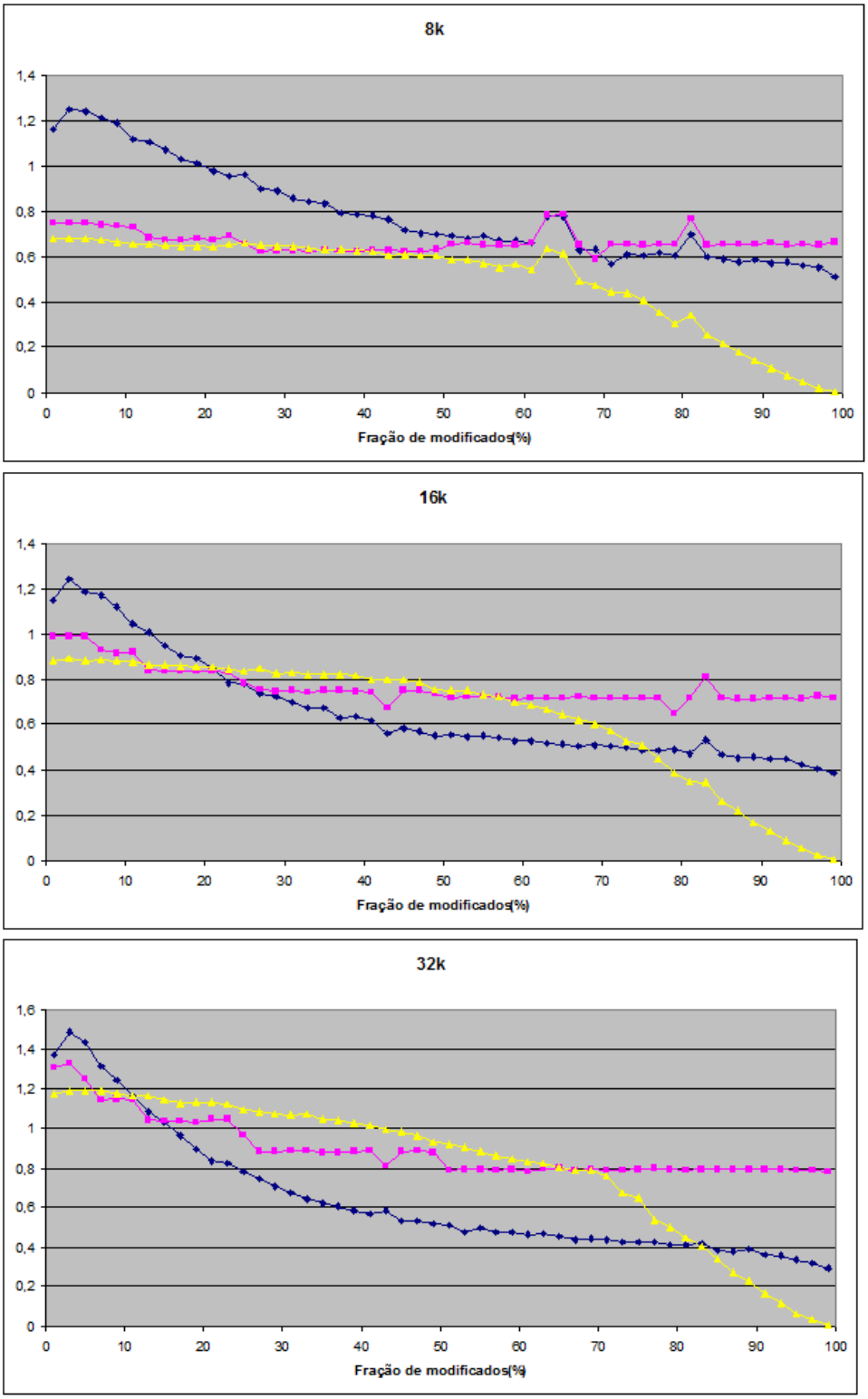

Figura B.5: Desempenhos relativos, 8k e 32k pares e chaves de 24 bits.

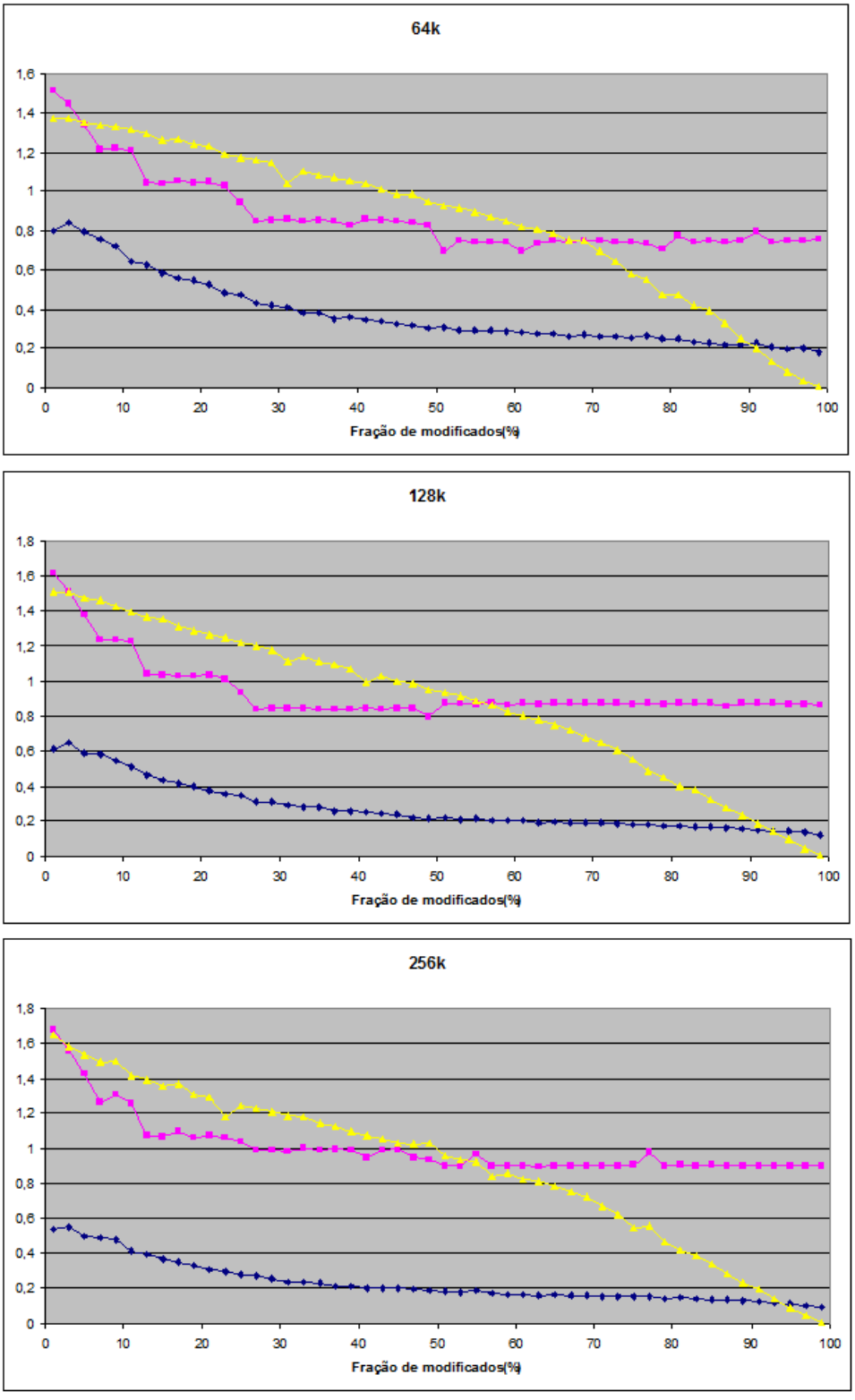

Figura B.6: Desempenhos relativos, 64k e 256k pares e chaves de 24 bits.

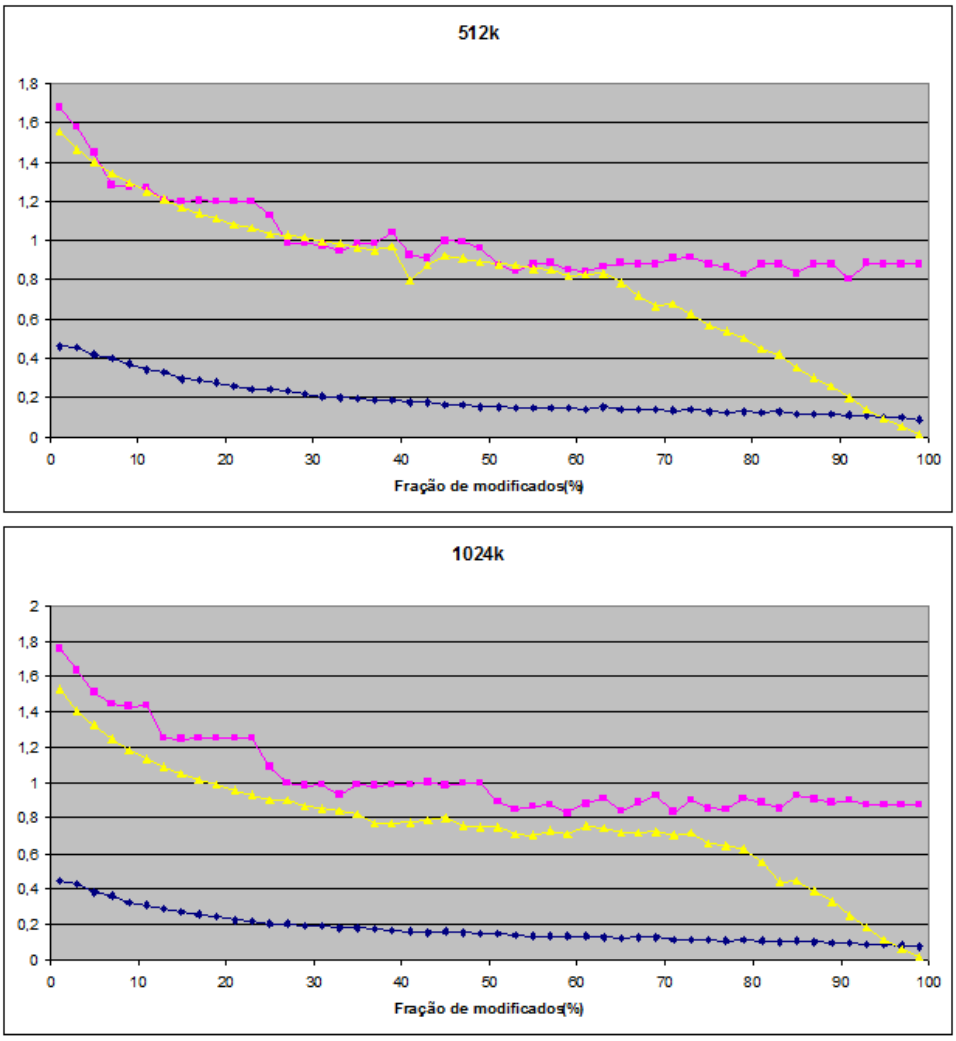

Figura B.7: Desempenhos relativos, 512k e 1024k pares e chaves de 24 bits.

B.3 Chaves de 28 bits

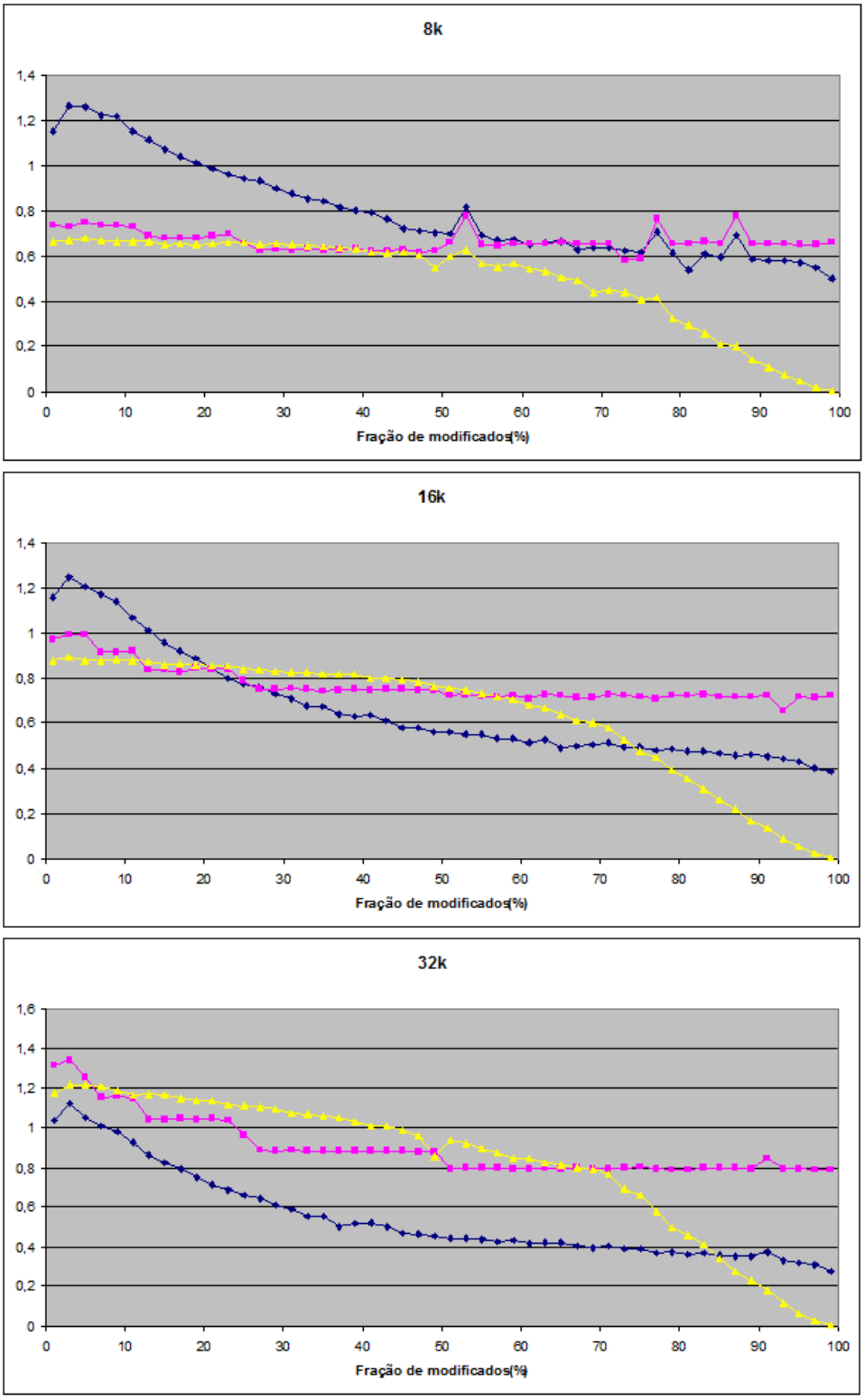

Figura B.8: Desempenhos relativos, 8k e 32k pares e chaves de 28 bits.

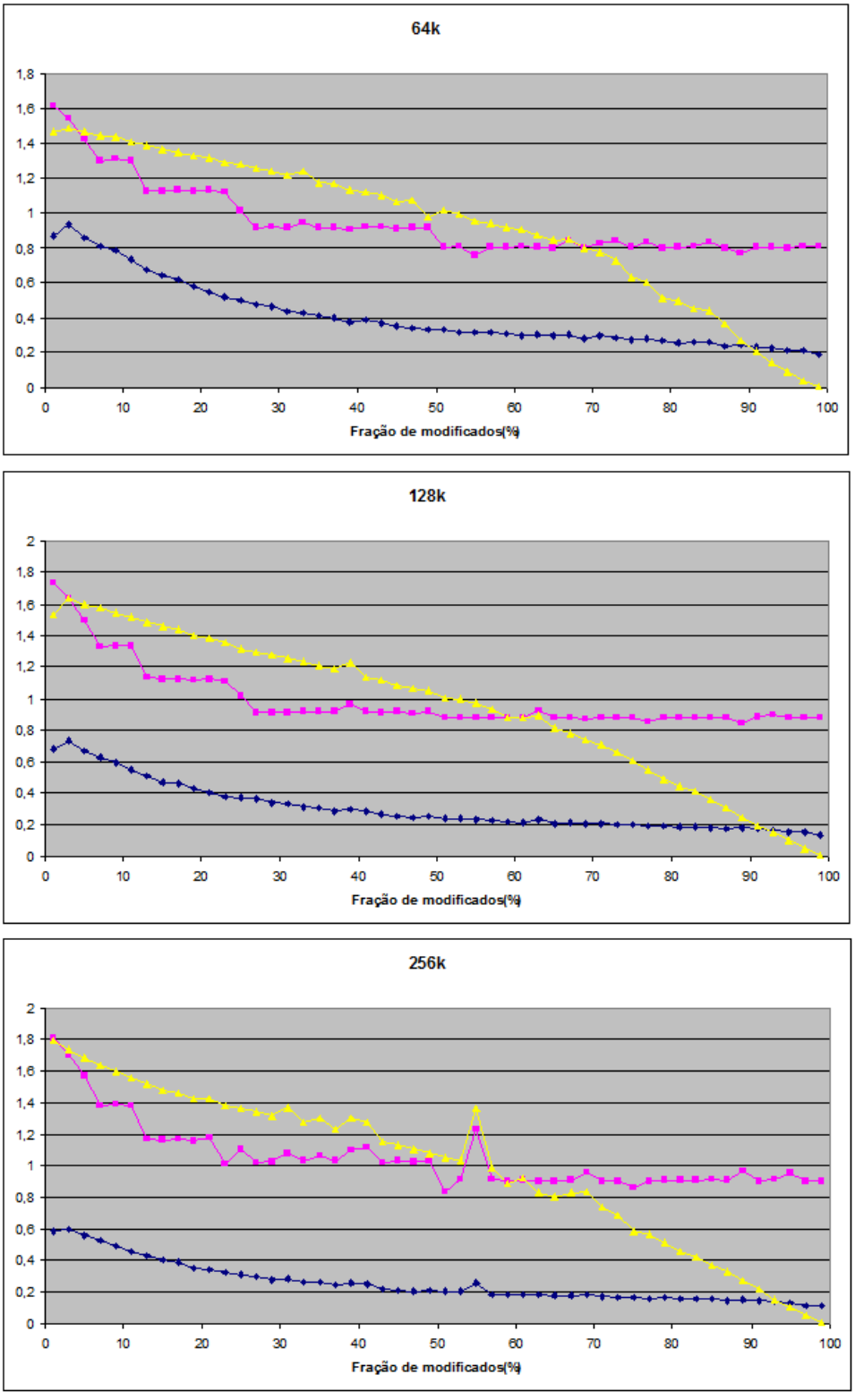

Figura B.9: Desempenhos relativos, 64k e 256k pares e chaves de 28 bits.

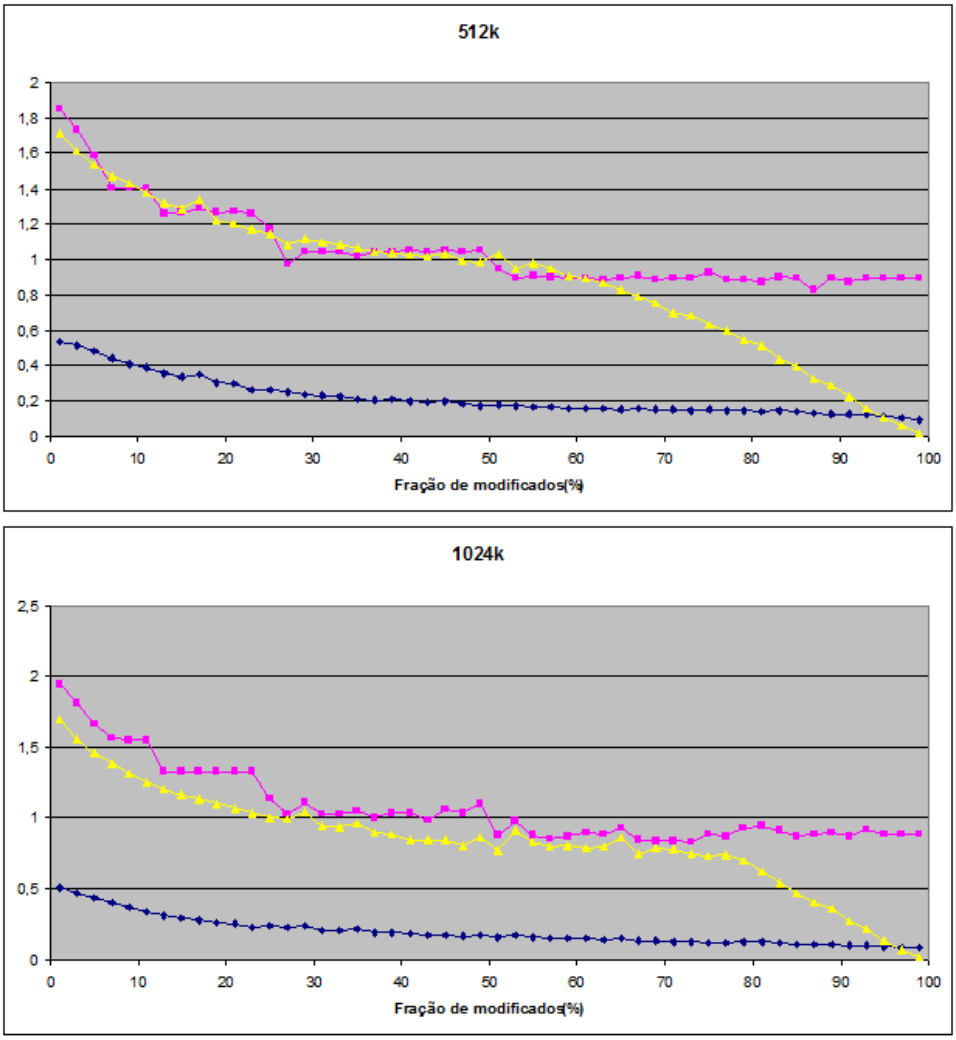

Figura B.10: Desempenhos relativos, 512k e 1024k pares e chaves de 28 bits.

B.4 Chaves de 32 bits

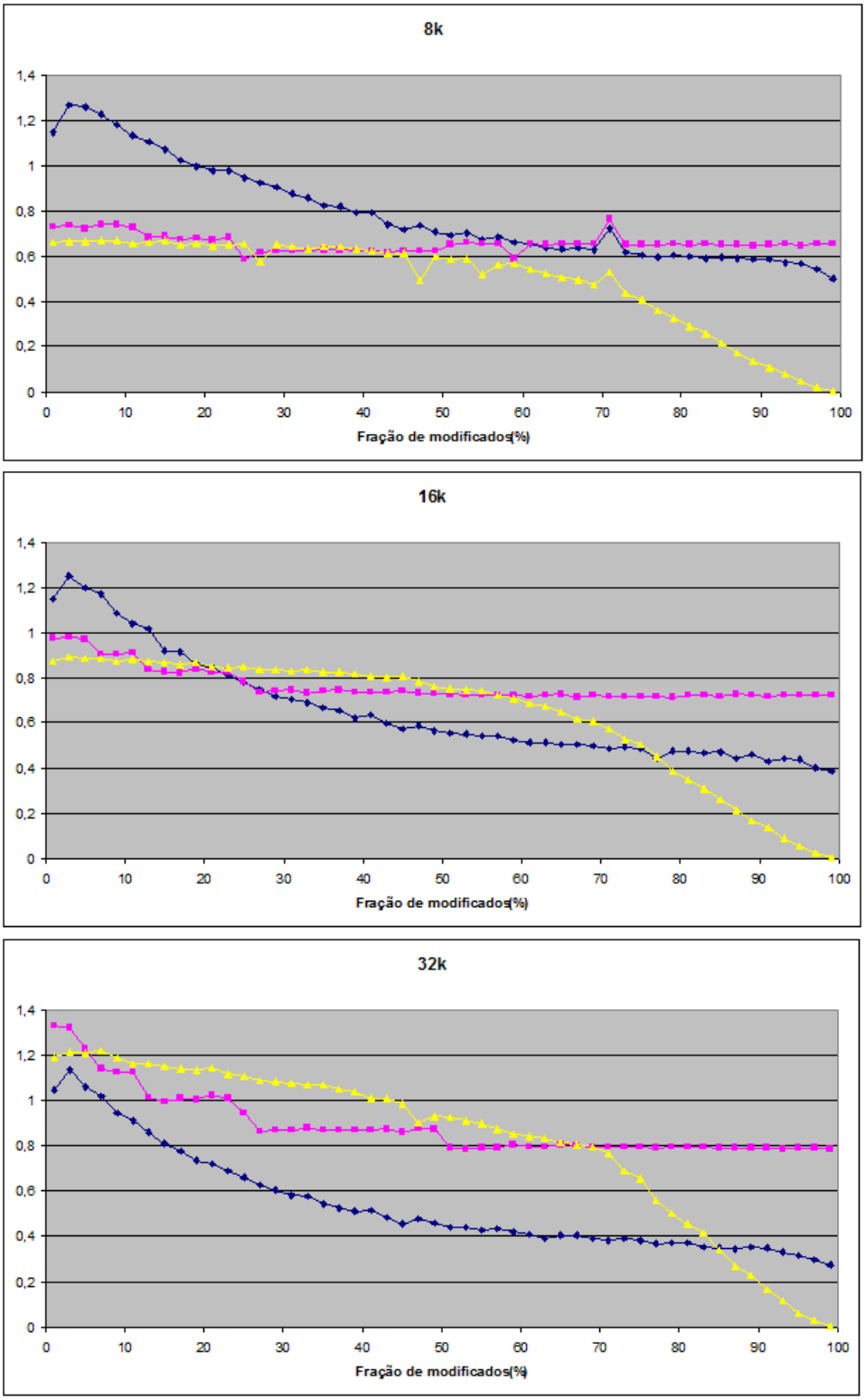

Figura B.11: Desempenhos relativos, 8k e 32k pares e chaves de 32 bits.

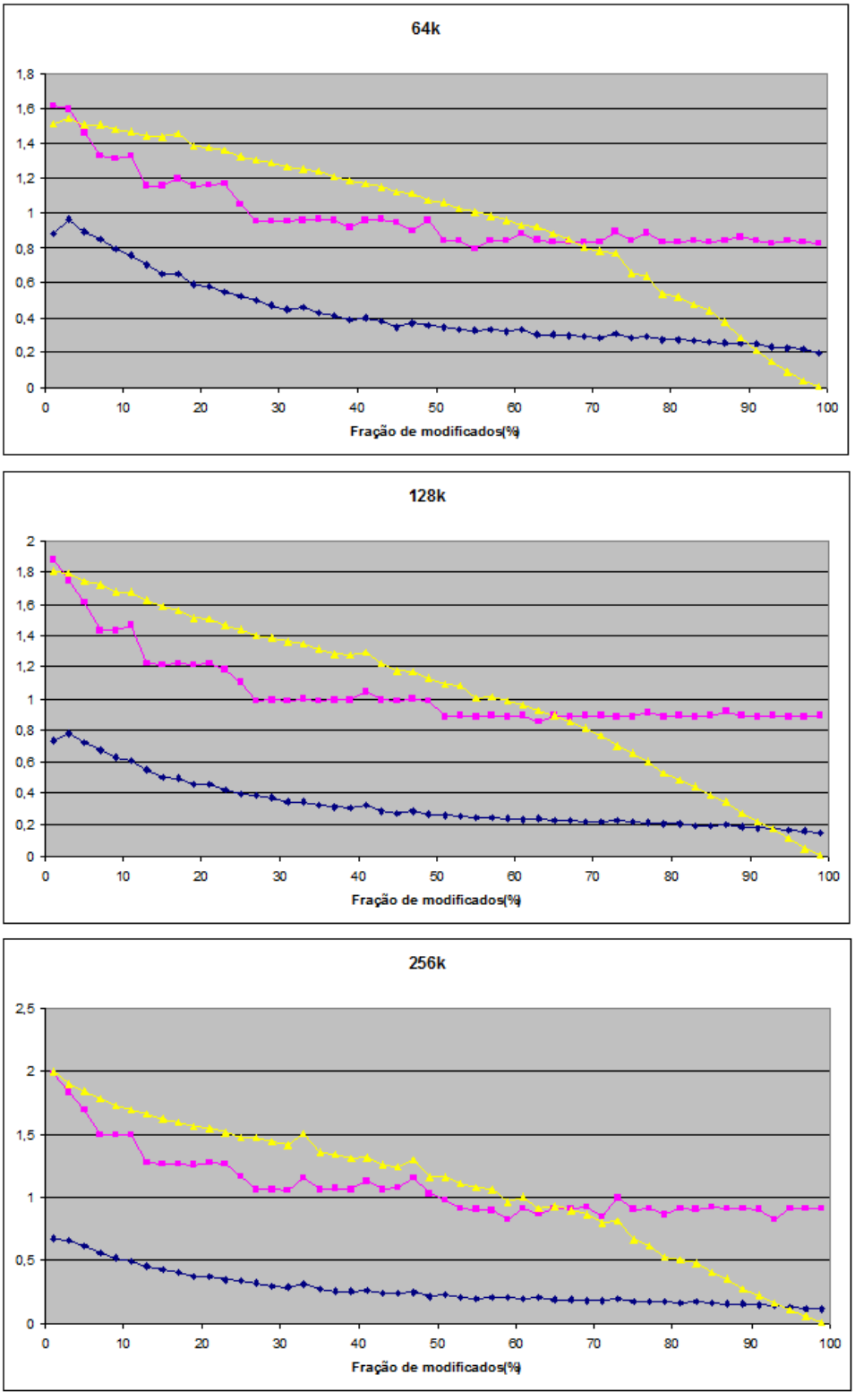

Figura B.12: Desempenhos relativos, 64k e 256k pares e chaves de 32 bits.

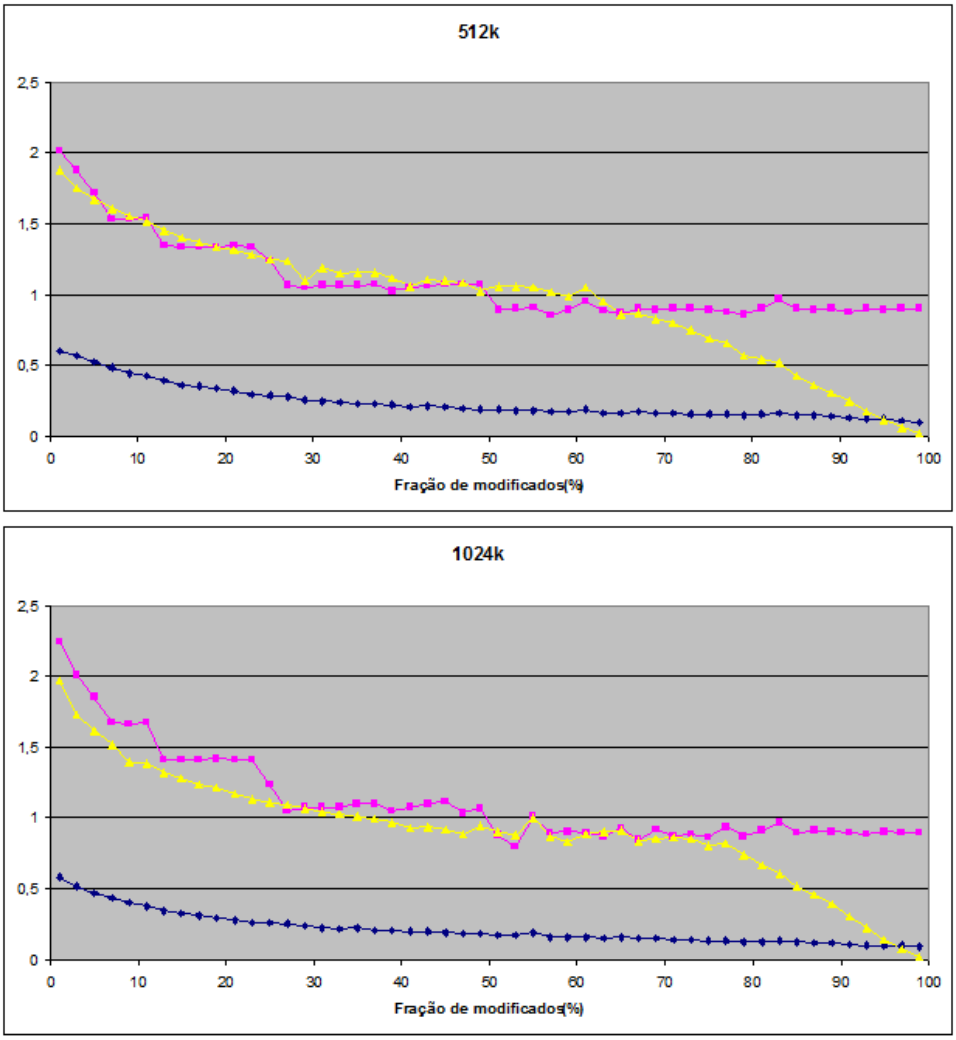

Figura B.13: Desempenhos relativos, 512k e 1024k pares e chaves de 32 bits.

Apêndice C

Avaliação dos métodos no sistema D.2 com chaves de 20, 24, 28 e 32 bits

As comparações a seguir foram realizadas segundo a explicação da seção 4.1, adotando a simbologia da figura C.1.

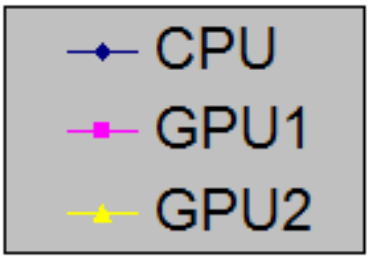

Figura C.1: Correspondência nos gráficos: CPU (Sec. 3.4), GPU1 (Subsec. 3.5.1) e GPU2 (Subsec. 3.5.2)

C.1 Chaves de 20 bits

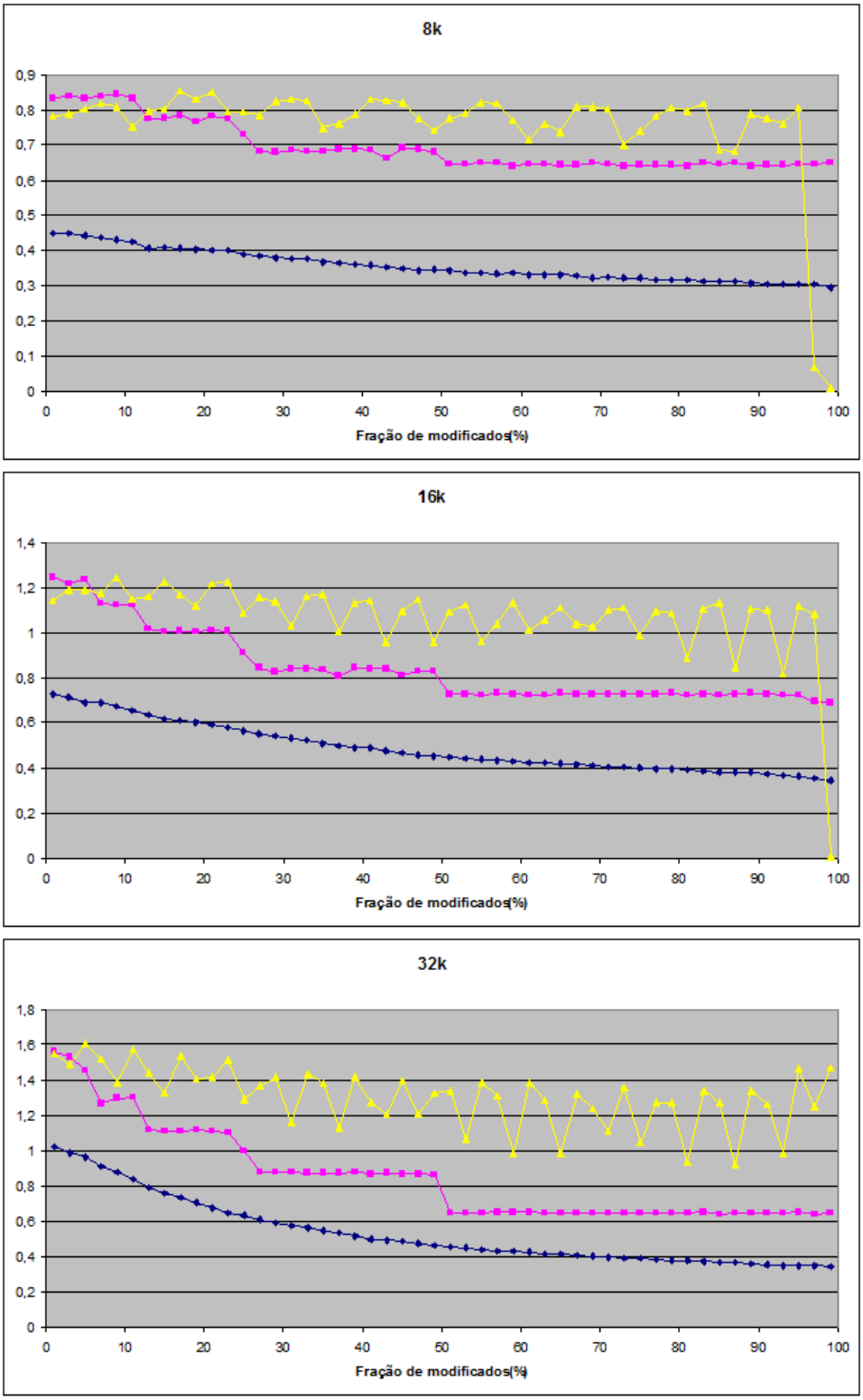

Figura C.2: Desempenhos relativos, 8k a 32k pares e chaves de 20 bits.

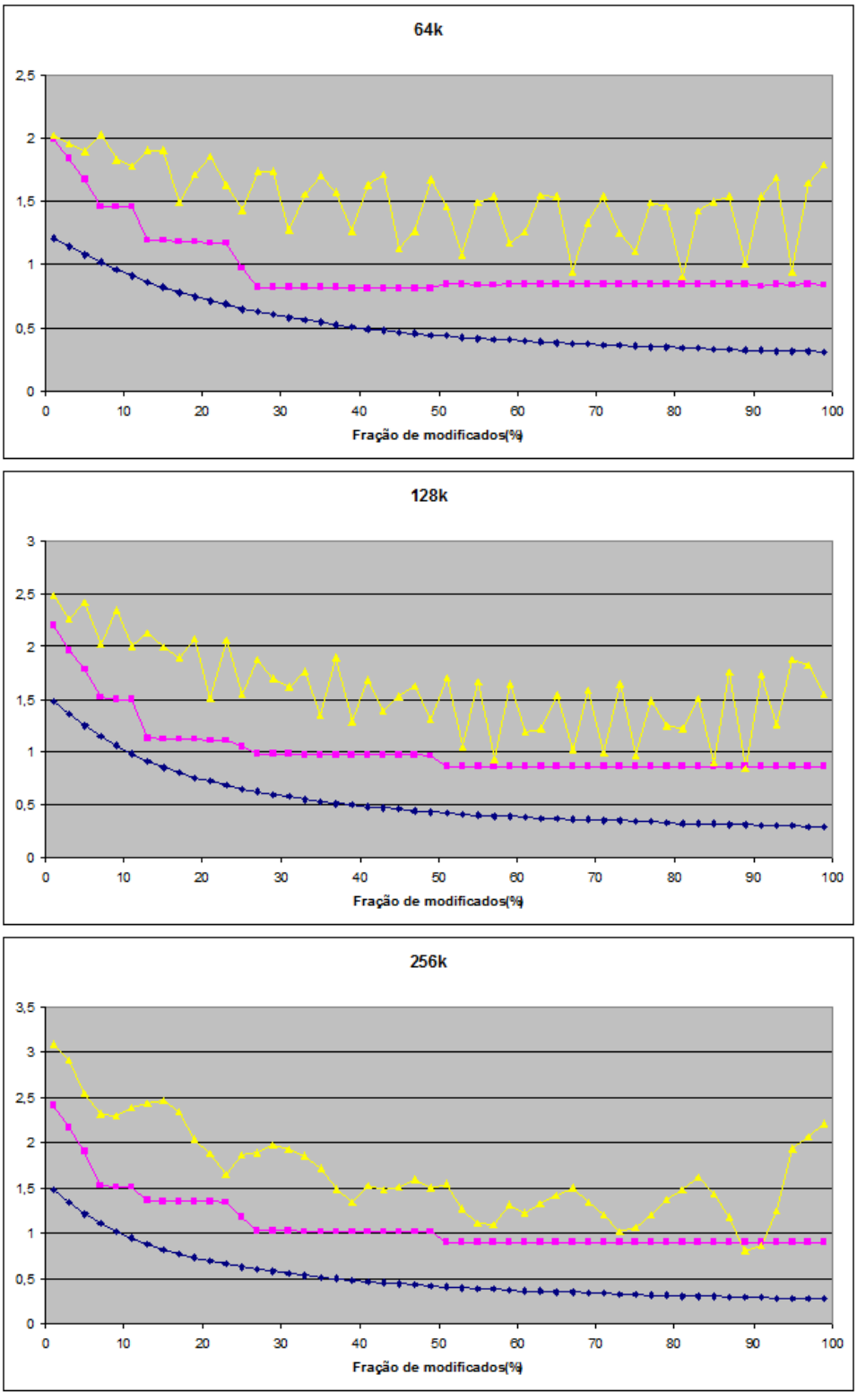

Figura C.3: Desempenhos relativos, 64k a 256k pares e chaves de 20 bits.

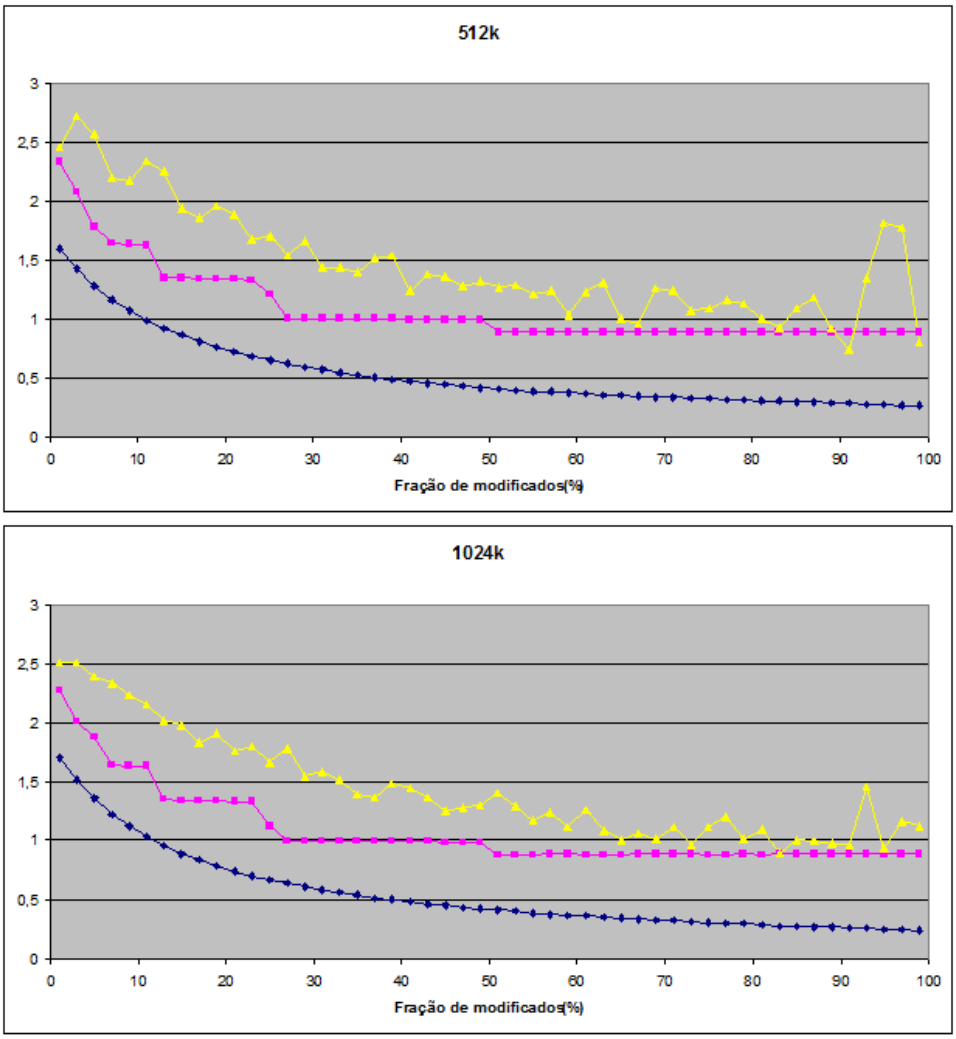

Figura C.4: Desempenhos relativos, 512k e 1024k pares e chaves de 20 bits.

C.2 Chaves de 24 bits

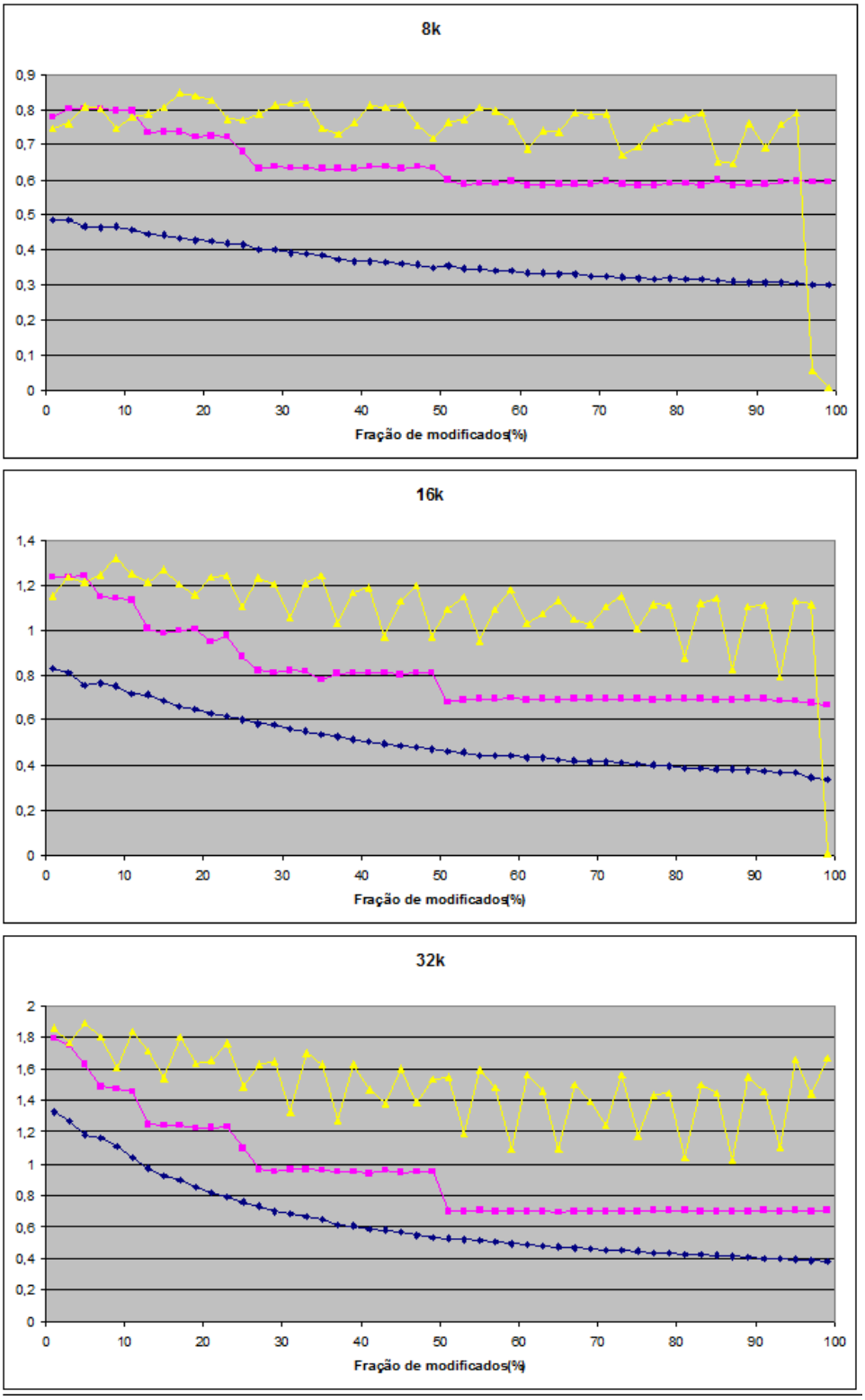

Figura C.5: Desempenhos relativos, 8k a 32k pares e chaves de 24 bits.

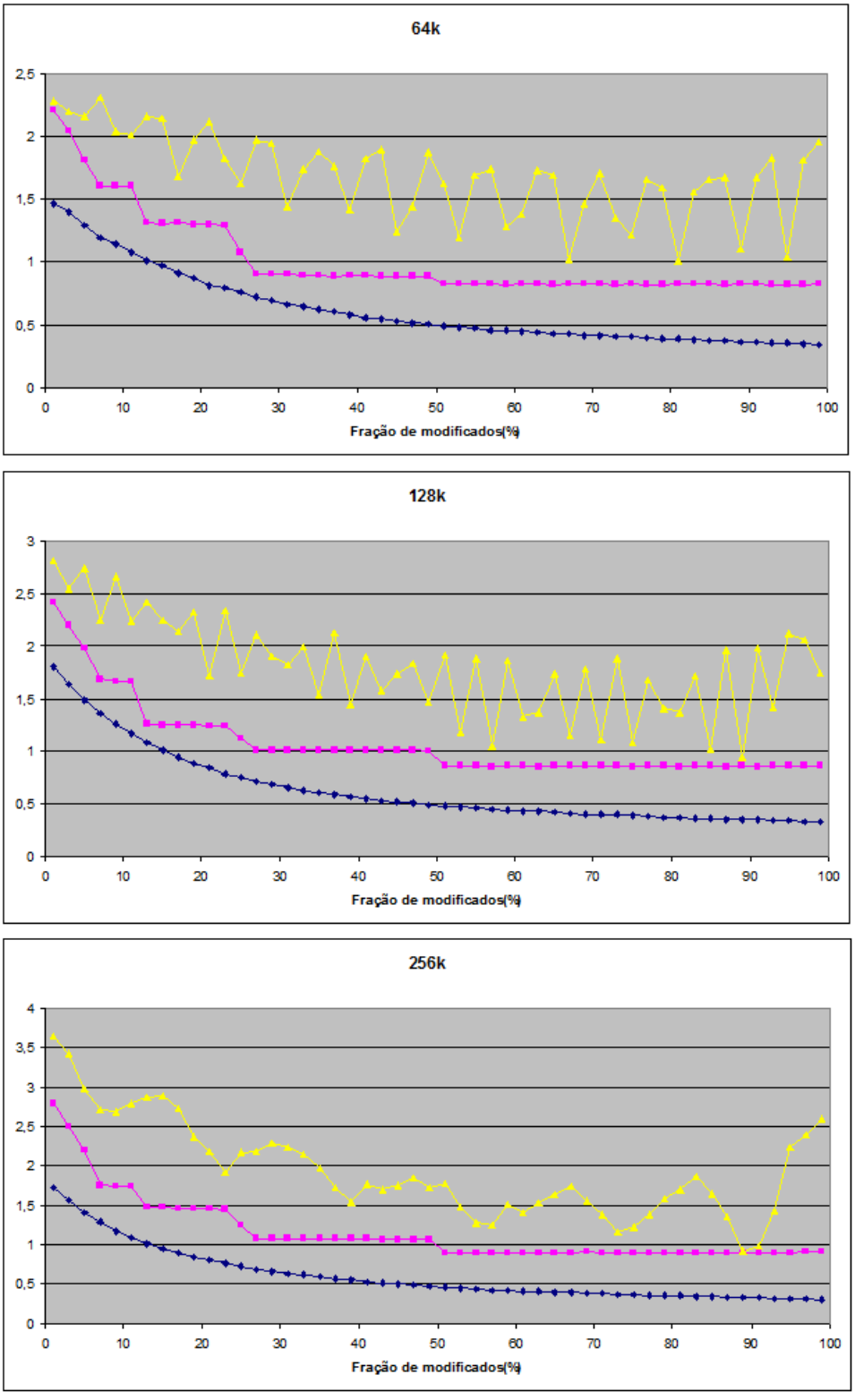

Figura C.6: Desempenhos relativos, 64k a 256k pares e chaves de 24 bits.

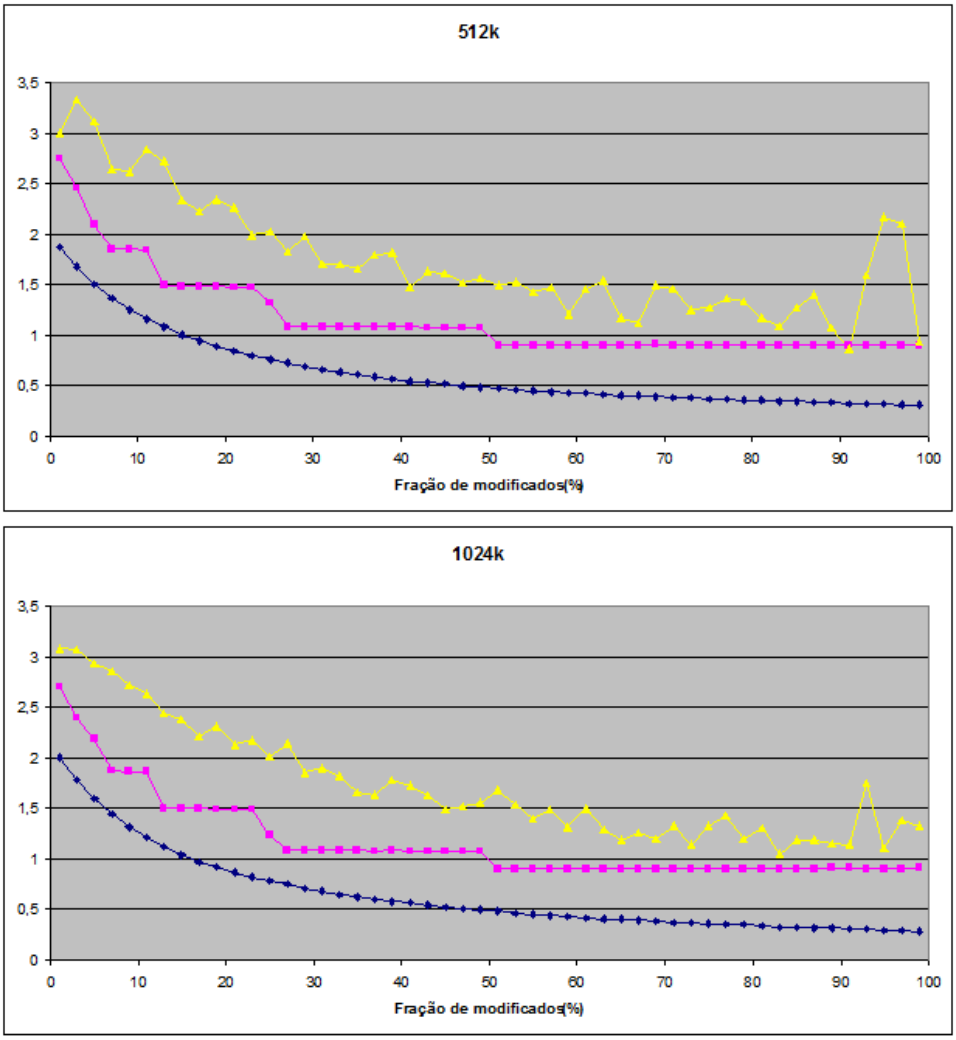

Figura C.7: Desempenhos relativos, 512k e 1024k pares e chaves de 24 bits.

C.3 Chaves de 28 bits

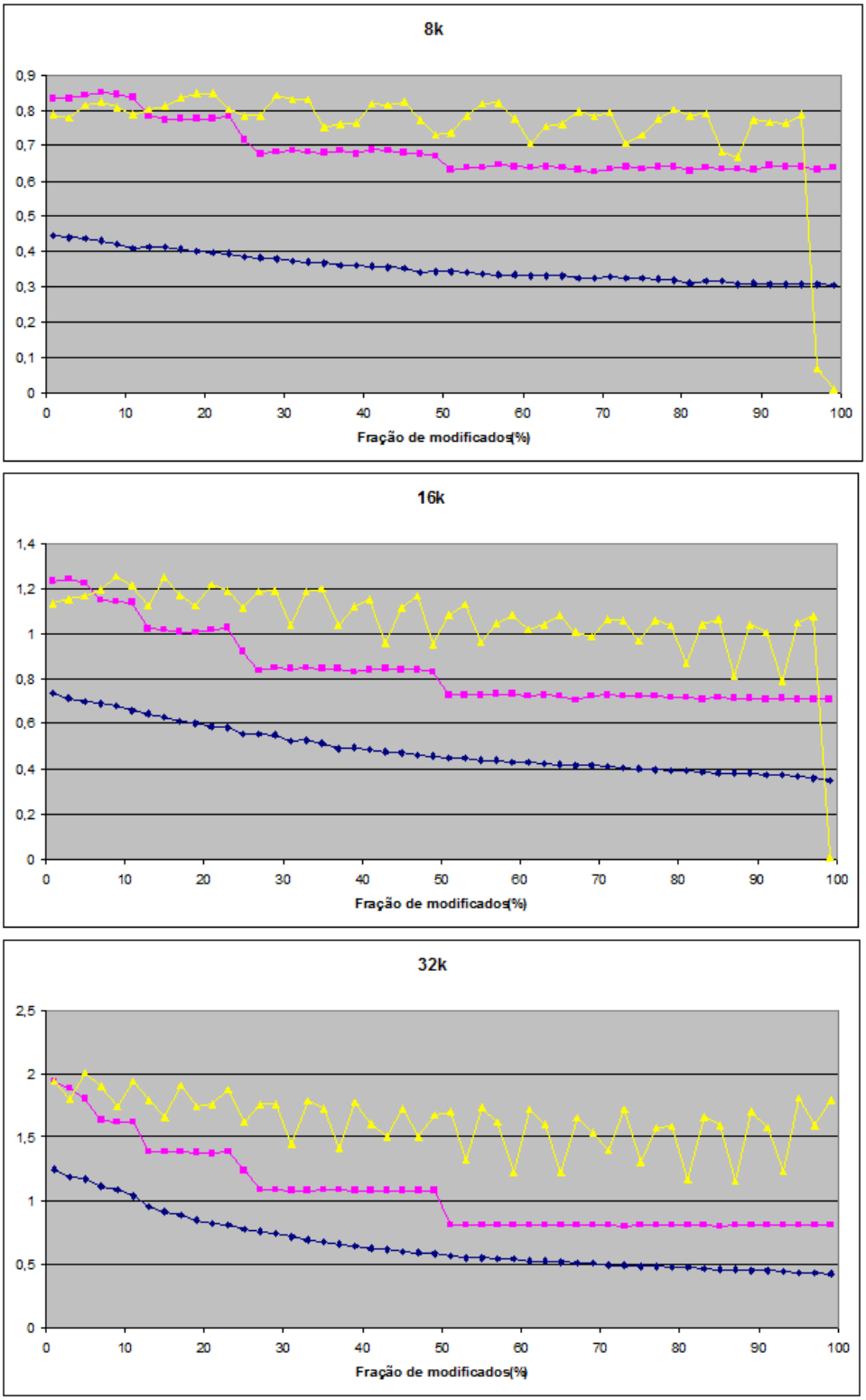

Figura C.8: Desempenhos relativos, 8k a 32k pares e chaves de 28 bits.

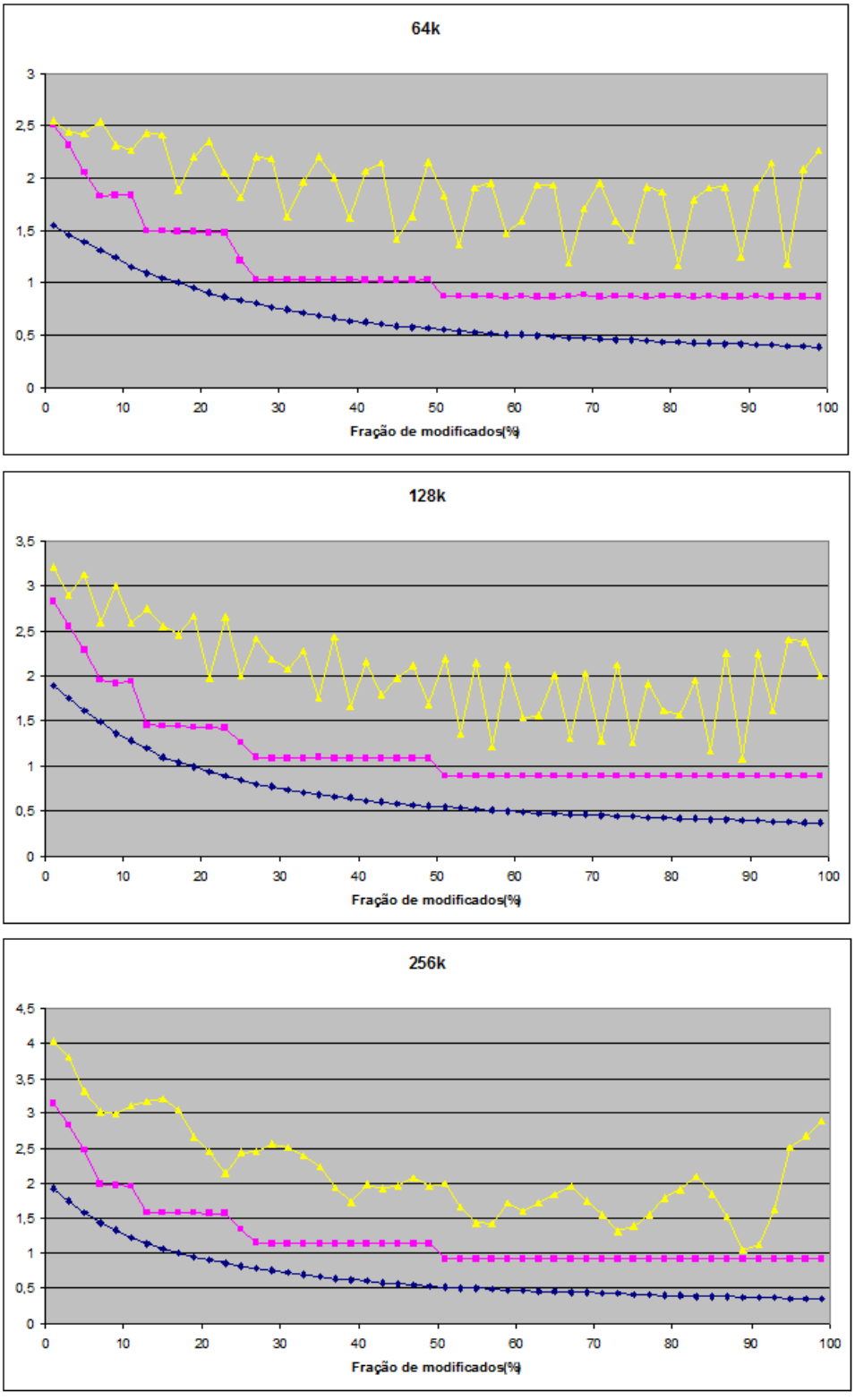

Figura C.9: Desempenhos relativos, 64k a 256k pares e chaves de 28 bits.

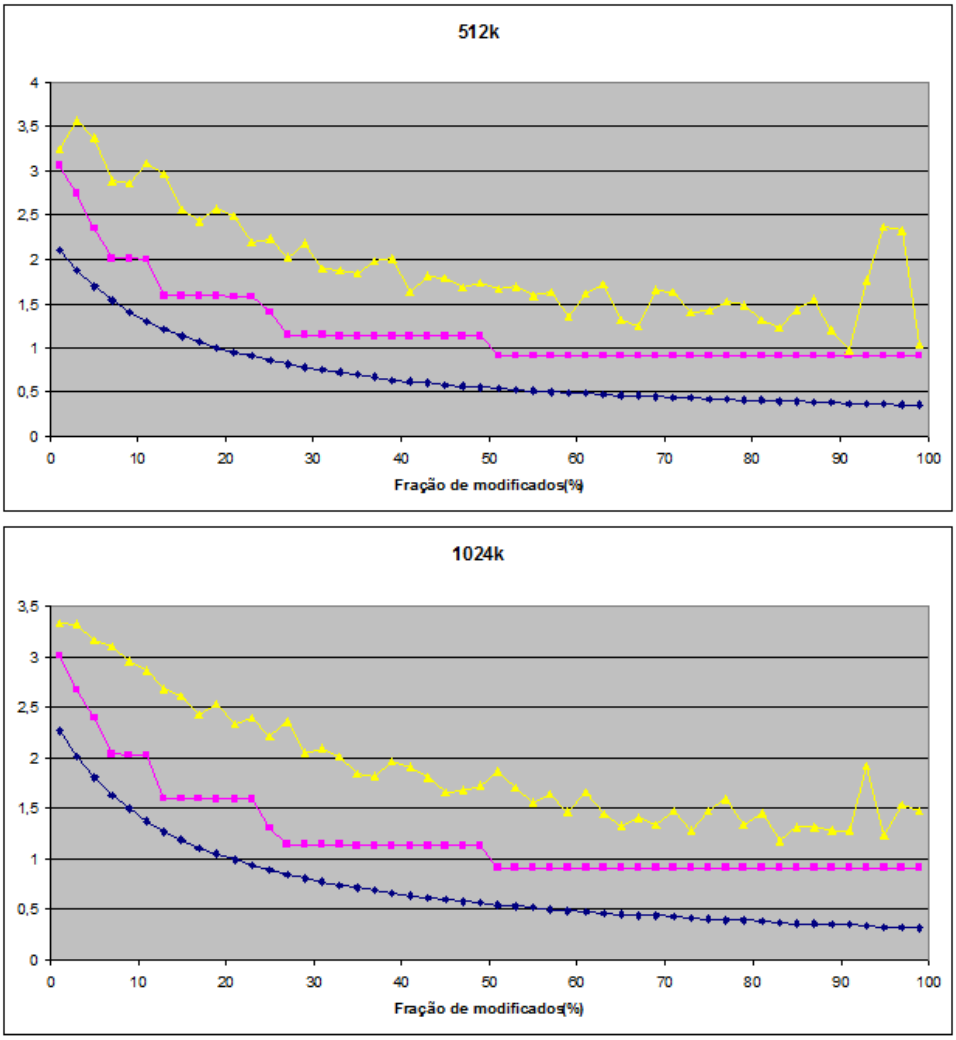

Figura C.10: Desempenhos relativos, 512k e 1024k pares e chaves de 28 bits.

C.4 Chaves de 32 bits

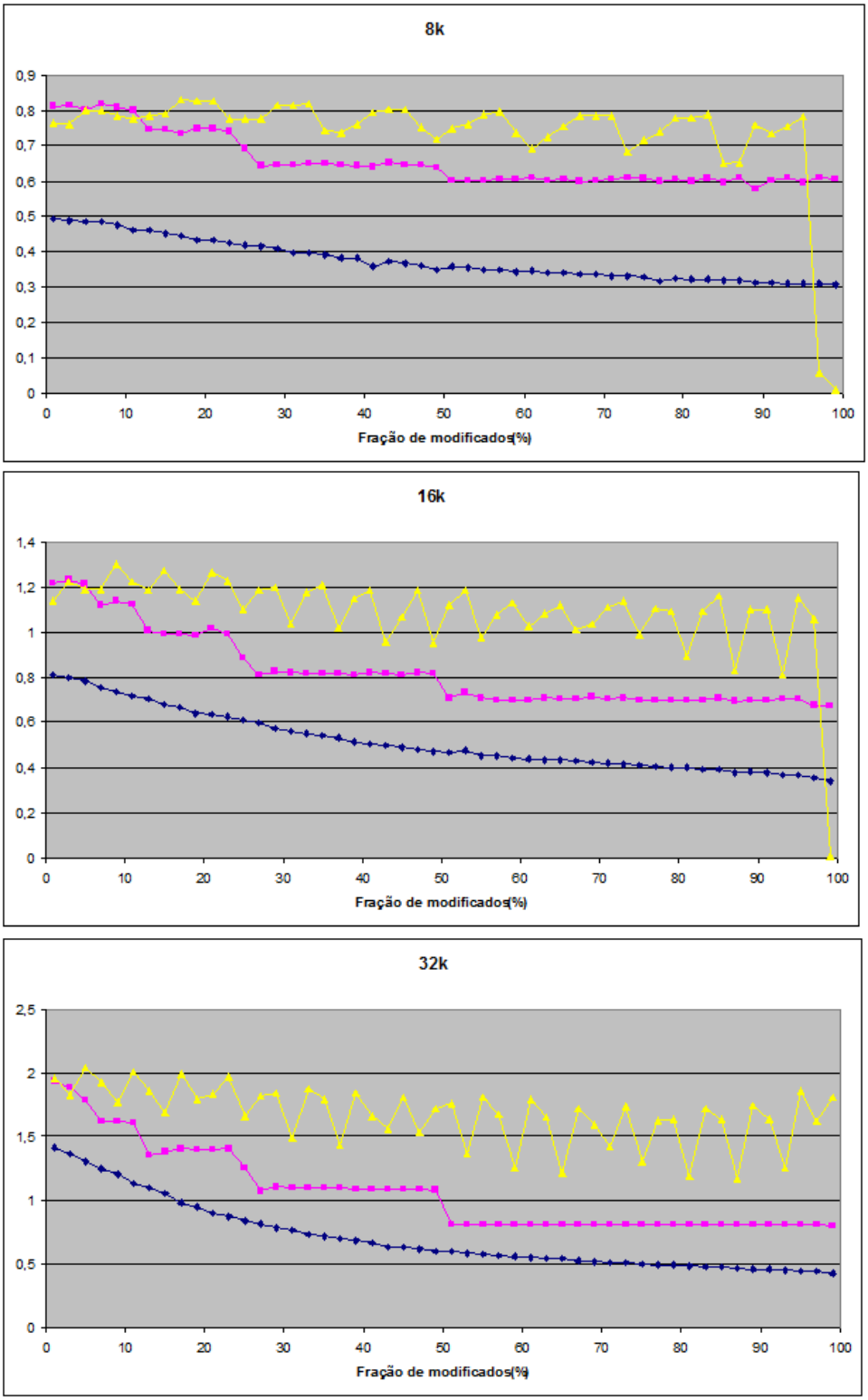

Figura C.11: Desempenhos relativos, 8k a 32k pares e chaves de 32 bits.

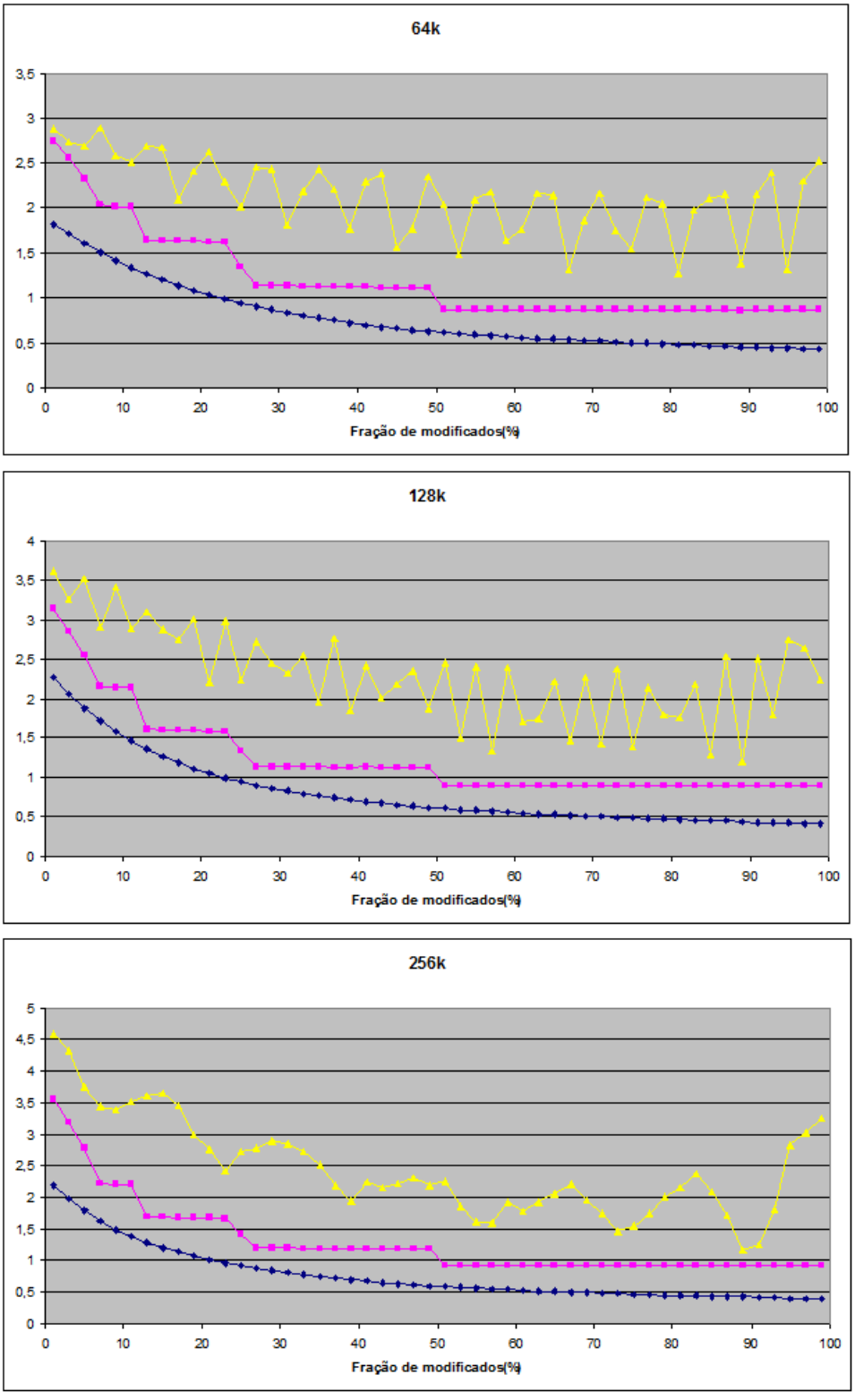

Figura C.12: Desempenhos relativos, 64k a 256k pares e chaves de 32 bits.

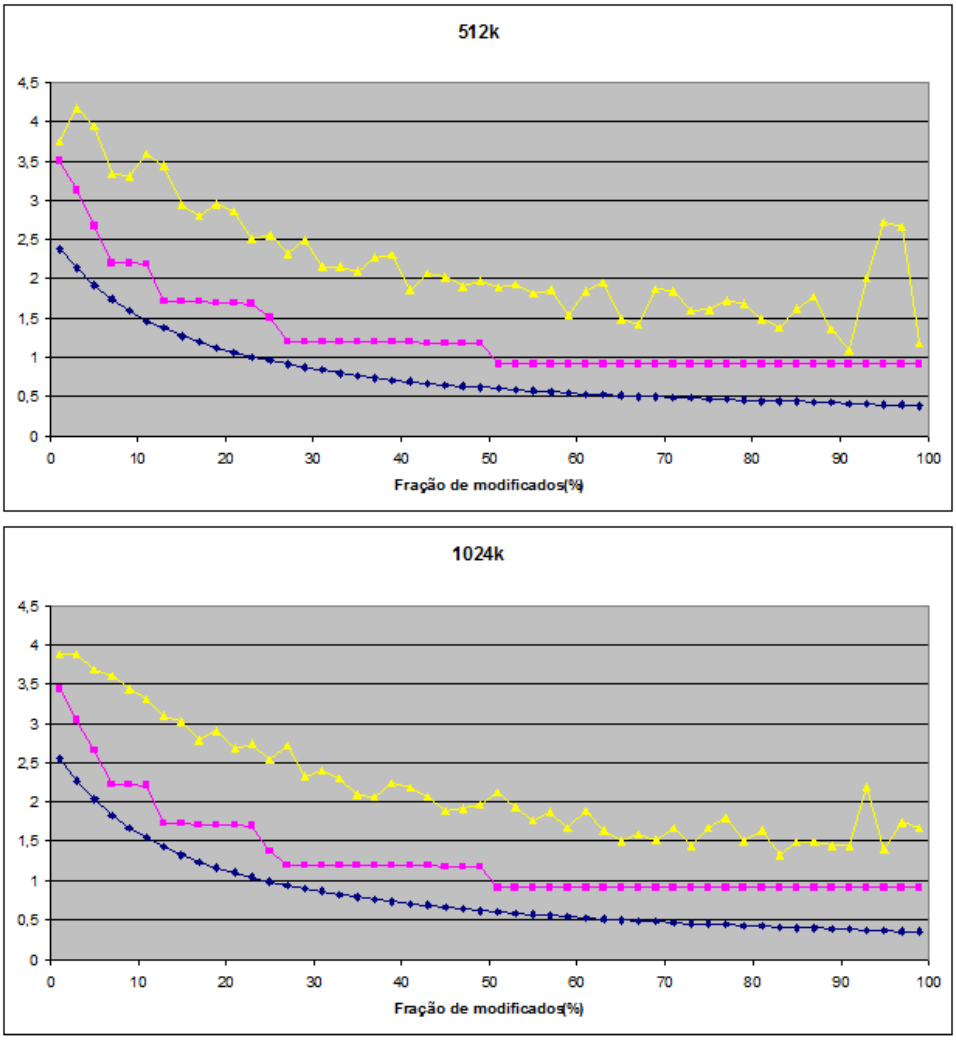

Figura C.13: Desempenhos relativos, 512k e 1024k pares e chaves de 32 bits.

Apêndice D

Sistemas utilizados nos testes

Foram dois os sistemas empregados nos *benchmarks* nesse trabalho, um *desktop* (Sistema 1) e um *notebook* (Sistema 2). As informações sobre os sistemas foram obtidas através do programa *Geeks3D GPU Caps Viewer 1.17.0*.

D.1 Sistema 1

D.1.1 System / CPU

- CPU Name: AMD Phenom(tm) II X4 940 Processor
- CPU Core Speed: 3000 MHz
- CPU logical cores: 4
- Family: 15 - Model: 4 - Stepping: 2
- Physical Memory Size: 4094 MB
- Operating System: Windows Vista 64-bit build 6002 [Service Pack 2]
- PhysX Version: 81029

D.1.2 Graphics Adapters / GPUs

- Current Display Mode: 1680x1050 @ 60 Hz - 32 bpp

- Num GPUs: 1
- GPU 1
- Name: ATI Radeon HD 5870
- GPU codename: Cypress
- Device ID: 1002-6898
- Subdevice ID: 1787-2289
- Driver: 8.951.0.0 - Catalyst 12.3 (3-8-2012)

D.2 Sistema 2

D.2.1 System / CPU

- CPU Name: Intel(R) Core(TM) i7-2630QM CPU @ 2.00GHz
- CPU Core Speed: 1995 MHz
- CPU logical cores: 8
- Family: 6 - Model: 10 - Stepping: 7
- Physical Memory Size: 8192 MB
- Operating System: Windows 7 64-bit build 7601 [Service Pack 1]
- PhysX Version: 9120209

D.2.2 Graphics Adapters / GPUs

- Current Display Mode: 1600x900 @ 60 Hz - 32 bpp

- Num GPUs: 2

- GPU 1
- Name: Intel HD Graphics 3000
- GPU codename: GT2(HD3000)
- Device ID: 8086- 116
- Subdevice ID: 1043-15E2
- Shader cores: 12
- Driver: 8.15.10.2509 (8-31-2011) - GL:ig4icd64.dll

- GPU 2
- Name: NVIDIA GeForce GT 540M
- GPU codename: GF108
- Device ID: 10DE- DF4
- Subdevice ID: 1043-15E2
- Driver: 8.15.10.2509 (R295.73)