Superando Limites no Multiparticionamento em GPU

Michel B. Cordeiro[1], **Wagner M. Nunan Zola** [1]

[1]Departamento de Informática – Universidade Federal do Paraná (UFPR)

michel.brasil.c@gmail.com, wagner@inf.ufpr.br

Abstract. *Multipartitioning is a general-purpose parallel primitive that reorganizes input data into contiguous bins (or buckets), where the function responsible for categorizing each element into a bin is defined by the programmer. Multipartitioning on GPUs, although considered as a parallel primitive in itself, has received little attention in the literature so far. State-of-the-art implementations focus on scenarios with a reduced number of bins (up to 256), also imposing the restriction that this number must be a power of 2. This work aims to present efficient implementations of Multipartitioning on GPUs, exploring different optimized strategies for varying numbers of bins, allowing the algorithm to select, at runtime, the most suitable level of granularity. The algorithm, named mrBin, is capable of processing large numbers of bins (up to 12,288), without imposing restrictions on this number. In other words, more bins, fewer restrictions. The experiments demonstrate that the algorithm surpasses the state of the art for bin counts greater than 64, achieving speedups of up to 2.2x.*

Resumo. *Multiparticionamento é uma primitiva paralela de propósito geral que reorganiza os dados de entrada em bins (ou buckets) contíguos, sendo a função responsável por categorizar cada elemento em um bin definida pelo programador. O Multiparticionamento em GPU, embora considerado uma primitiva paralela em si, recebeu pouca atenção até o momento na literatura. As implementações estado da arte concentram-se em cenários com número reduzido de bins (até 256), impondo a restrição de que essa quantidade seja potência de 2. Este trabalho tem como objetivo apresentar implementações eficientes de Multiparticionamento em GPU, explorando diferentes estratégias otimizadas para diferentes quantidades de bins, permitindo que o algoritmo selecione, em tempo de execução, o nível de granularidade mais adequado. O algoritmo, denominado mrBin, é capaz de processar grandes quantidades de bins (até 12.288), sem impor restrições sobre esse número. Ou seja, mais bins, menos restrições. Nos experimentos realizados, o algoritmo mostrou-se capaz de superar o estado da arte para quantidades de bins maiores que 64, alcançando aceleração de até 2,2 vezes.*

1. Introdução

O multiparticionamento, ou *multisplit*, é um algoritmo paralelo cujo objetivo é reorganizar um vetor de dados de entrada em *bins* (ou *buckets*) contíguos na memória, utilizando uma função fornecida pelo programador para categorizar cada elemento em sua respectiva partição. Formalmente, o *multisplit* pode ser definido da seguinte forma: Dado um conjunto de elementos $S = \{x_1, x_2, \ldots, x_n\}$ e uma função de categorização $f : S \rightarrow \{0, 1, \ldots, k-1\}$, o objetivo é particionar S em k subconjuntos disjuntos $B_0, B_1, \ldots, B_{k-1}$, tais que $B_i = \{x \in S \mid f(x) = i\}$, para $i \in \{0, 1, \ldots, k-1\}$.

O multiparticionamento é uma primitiva fundamental na programação em GPU, sendo utilizado em diversos algoritmos, como na construção de tabelas *hash* [Alcantara et al. 2009], [Lessley and Childs 2020], [Jünger et al. 2020], em implementações do *Radix Sort* [Merrill and Grimshaw 2010], [Stehle and Jacobsen 2017], e na construção de *KD-trees* [Choi et al. 2010]. Dessa forma, sua implementação eficiente é crucial para garantir alto desempenho, uma vez que essa primitiva é utilizada repetidamente durante a execução dessas aplicações.

No entanto, apesar de sua relevância, o multiparticionamento em GPU, considerado como uma primitiva paralela em si, tem recebido relativamente pouca atenção na literatura. Uma implementação eficiente, denominada *GPU multisplit*, foi apresentada por Ashkiani et al. [Ashkiani et al. 2017]. Embora represente o estado da arte, essa abordagem é restrita a cenários com número reduzido de *bins*, impondo limitações como a exigência de que a quantidade de *bins* seja uma potência de dois e não ultrapasse 256. Sendo assim, as restrições da implementação de Ashkiani et al. tornam-na pouco prática em cenários gerais.

Diante disso, o presente trabalho propõe uma alternativa eficiente de multiparticionamento em GPU, capaz de suportar mais de 12 mil partições e sem a limitação de que o número de *bins* seja uma potência de dois. Com esse objetivo, foram desenvolvidas quatro estratégias distintas para tratar o problema e, a partir de uma análise comparativa, selecionaram-se aquelas que apresentaram melhor desempenho em cenários específicos. Isso possibilitou o desenvolvimento do algoritmo adaptativo denominado *mrBins* (*Multi-partition with a wider-Range of Bins*), cuja principal característica é a escolha dinâmica da granularidade de processamento, adaptando-se em tempo de execução. As principais contribuições deste trabalho podem ser resumidas da seguinte forma:

(i) **Análise comparativa de quatro estratégias de multisplit em GPU:** foram investigadas diferentes abordagens, identificando-se suas vantagens e limitações.
(ii) **Desenvolvimento do algoritmo adaptativo *mrBins*:** foi desenvolvido um algoritmo capaz de selecionar dinamicamente, em tempo de execução, a estratégia mais adequada, garantindo eficiência em diversos cenários.
(iii) **Suporte a milhares de *bins* sem restrições rígidas (Mais *Bins*, Menos Restrições):** em comparação com o estado da arte, o *mrBins* é capaz de processar mais de 12 mil *bins*, sem a limitação de que o número de *bins* seja uma potência de dois.
(iv) **Resultados experimentais de alto desempenho:** foi demonstrado que o algoritmo *mrBins* supera a implementação estado da arte em cenários com mais de 64 *bins*, alcançando acelerações superiores a 2 vezes.

O restante do artigo está organizado da seguinte forma: a Seção 2 apresenta os trabalhos relacionados; na Seção 3, são apresentados os fundamentos teóricos; na Seção 4, são detalhadas as estratégias de implementação; na Seção 5, é descrita a metodologia dos experimentos; na Seção 6, é realizada a análise dos resultados; e, finalmente, na Seção 7, são apresentadas as conclusões deste estudo.

2. Trabalhos Relacionados

O artigo de Ashkiani et al. [Ashkiani et al. 2016], intitulado *GPU Multisplit*, apresenta uma abordagem para o multiparticionamento paralelo em GPUs, com foco na otimização

de algoritmos que requerem particionamento de dados, como o *Radix Sort*. A proposta busca melhorar o desempenho ao minimizar a divergência de *threads* e otimizar o acesso à memória, explorando as características das arquiteturas de GPU. Por meio de uma análise detalhada dos padrões de acesso e das dependências de memória, os autores introduzem técnicas para reduzir a latência e melhorar a coalescência dos acessos à memória global. Apesar de sua eficácia em cenários de menor escala, o *GPU Multisplit* enfrenta limitações quando o número de *bins* é grande, sendo mais adequado para problemas com uma quantidade reduzida de *bins* (até 256), e exigindo que esse número seja uma potência de dois.

Em 2017, os autores expandiram esse estudo, propondo melhorias para o *GPU Multisplit* [Ashkiani et al. 2017]. As novas abordagens resultaram em desempenho superior em tarefas de ordenação e particionamento, com destaque para otimizações no acesso à memória e redução da latência. Essas melhorias tornaram o algoritmo mais eficiente em aplicações como a construção de tabelas *hash* e outras operações de ordenação, embora ainda permanecessem restrições quanto ao número de *bins*.

O trabalho de Cordeiro e Nunan Zola [Cordeiro and Nunan Zola 2025] propõe um algoritmo de multiparticionamento em GPU, apresentando duas estratégias distintas: uma voltada para cenários com grande número de *bins* e outra para casos com menor quantidade. A versão destinada a poucos *bins* foi comparada ao algoritmo *GPU Multisplit* [Ashkiani et al. 2017], alcançando uma aceleração de até 83%. Esse artigo serviu como base para o desenvolvimento do presente trabalho, e ambas as estratégias serão discutidas em detalhe na seção de implementação.

De forma complementar, os autores também exploraram o multiparticionamento no contexto de computação paralela em CPU, propondo em [Cordeiro et al. 2025b] um algoritmo paralelo eficiente para processadores *multicore*. O algoritmo utiliza técnicas de balanceamento de carga e otimização de acesso à memória para maximizar a utilização dos núcleos de processamento e minimizar contenção de recursos. Além disso, demonstra boa escalabilidade, mantendo desempenho eficiente à medida que o número de núcleos aumenta, o que o torna adequado para aplicações de grande escala.

A proposta também demonstrou ser capaz de acelerar operações de ordenação chave-valor, conforme apresentado no trabalho "*Algoritmo Paralelo e Distribuído para Ordenação Chave-Valor*" [Cordeiro et al. 2025a]. Os experimentos realizados mostraram que a solução alcançou aceleração de até 3,3 vezes em relação ao método padrão `std::par` com a biblioteca *Intel Threading Building Blocks* (TBB) [Reinders 2007]. Esses resultados demonstram como uma implementação eficiente de multiparticionamento pode contribuir significativamente para a aceleração de outros processos. Embora os estudos apresentados abordem o problema de multiparticionamento em arquiteturas diferentes, e apresentam características distintas, esses trabalhos servem como base e motivação para o desenvolvimento do *mrBins*.

3. Fundamentos Teóricos

As GPUs possuem grande capacidade de acelerar algoritmos, devido ao seu processamento massivamente paralelo. No entanto, para obter desempenho, é importante compreender alguns conceitos. As GPUs têm uma arquitetura baseada no modelo de execução SIMT (*Single Instruction, Multiple Threads*), que organiza as *threads* em *warps*. Os *warps*, que geralmente consistem em 32 *threads* no modelo CUDA, executam instruções

simultaneamente. Isso significa que divergências de controle podem ocorrer quando *threads* de um mesmo *warp* precisam seguir caminhos de execução diferentes, reduzindo a eficiência do processamento paralelo.

Outro aspecto importante está relacionado à hierarquia de memória das GPUs, que é dividida em diferentes tipos. A memória global é a memória principal da GPU, acessível por todas as *threads*, porém apresenta uma latência relativamente alta. Por outro lado, a memória compartilhada (*shared memory*) possui menor latência e maior velocidade, mas capacidade limitada. Ela é visível apenas para as *threads* pertencentes a um mesmo bloco de *threads*. A utilização eficiente dessa memória permite que as *threads* troquem dados rapidamente, reduzindo a necessidade de acessos à memória global e aumentando o desempenho. Além disso, o acesso coalescido à memória global é essencial para garantir eficiência. O acesso coalescido consiste em organizar os acessos de modo que várias *threads* de um *warp* acessem posições contíguas na memória global. Quando os acessos são coalescidos, várias leituras ou escritas podem ser agrupadas em uma única transação, reduzindo a latência. Caso contrário, o desempenho do programa é comprometido.

Neste contexto, embora a versão sequencial do algoritmo de multiparticionamento consista em percorrer os dados de forma linear, aplicando a função de particionamento e distribuindo os elementos nos *bins* correspondentes, a versão paralela em GPU apresenta desafios adicionais relacionados à coordenação das threads e ao acesso à memória global. Os principais desafios incluem:

- **Granularização do problema:** Dividir a carga de trabalho de forma que seja balanceada entre as *threads*, evitando *threads* ociosas e maximizando o uso da GPU. Uma granularização inadequada pode impactar negativamente o desempenho.
- **Comunicação entre threads:** Minimizar a troca de informações entre *threads* para reduzir latência de memória ou sincronização. Em grandes volumes de dados, a comunicação excessiva pode comprometer o desempenho.
- **Uso eficiente da memória:** Garantir acessos coalescidos à memória global e explorar *shared memory* para reduzir latência e melhorar a eficiência.
- **Divergência de *warp*:** Evitar a divergência no fluxo de execução de *threads* dentro de um *warp*, que podem resultar em execução serializada e degradação de desempenho.

4. Descrição do Algoritmo

O algoritmo proposto é dividido em duas etapas. Inicialmente, cada bloco de *threads* percorre sua fatia dos dados de entrada, categorizando os elementos e construindo histogramas locais que contabilizam quantos elementos pertencem a cada *bin* em sua porção. A divisão dos dados é feita de forma que cada *thread* processe um subconjunto contíguo de elementos, garantindo balanceamento de carga entre as *threads*. Os histogramas locais são armazenados em *shared memory*, evitando acessos concorrentes à memória global. Em seguida, os totais por *bin* de todos os blocos são combinados em um vetor global, sobre o qual é aplicada uma soma de prefixos para determinar os *offsets* de cada partição na memória global. Essa abordagem permite que cada *thread* saiba exatamente onde escrever seus elementos na saída, realizando a alocação de forma paralela e eficiente. Em seguida, os histogramas parciais são combinados para gerar um histograma global de toda a entrada e aplicar uma operação de *prefix-sum*, permitindo calcular os *offsets* globais de

cada *bin*. Além disso, também é realizado um *prefix-sum* "por coluna" nos histogramas locais, determinando os *offsets* locais de cada bloco. Com esses *offsets*, cada bloco de *threads* sabe exatamente para onde enviar seus elementos no vetor de saída.

Na segunda etapa, os elementos são redistribuídos para os *bins* de destino, produzindo o vetor de saída e garantindo que todos os elementos de um mesmo *bin* fiquem contíguos na memória. A realização de *prefix-sums* "por coluna" na primeira etapa, embora adicione um pequeno custo ao processamento inicial, permite que a segunda etapa seja realizada sem operações atômicas, evitando a serialização das *threads* e, assim, contribuindo para o bom desempenho global dos *kernels* de multiparticionamento que fazem parte do *mrBins*.

A primeira etapa é comum em todas as abordagens discutidas. Para a segunda etapa, quatro estratégias foram consideradas:

(i) ***large:*** A primeira abordagem, denominada *large*, aplica diretamente a função de categorização e grava cada elemento no seu respectivo *bin* no vetor final. Nessa versão, a *shared memory* é utilizada apenas para construir os histogramas e realizar o *scan* na etapa inicial do algoritmo, permitindo o processamento de um número elevado de *bins*. No entanto, devido à ausência de organização e "bufferização" nas operações de escrita, essa versão realiza muitas escritas não coalescidas em memória global, o que pode resultar em queda de desempenho.

(ii) ***buffer-per-bin:*** A segunda versão, denominada *buffer-per-bin*, utiliza *buffers* na *shared memory* para armazenar temporariamente os elementos de cada *bin*, gravando-os na memória global apenas quando o *buffer* atinge 32 elementos. Dessa forma, um *warp* inteiro pode realizar a escrita de maneira coalescida, liberando espaço para que outras *threads* armazenem seus elementos. A principal otimização é que a sobrecarga de manter os *buffers* é relativamente pequena e não impõe restrições quanto ao número de *bins*, diferentemente da limitação de potência de dois observada no algoritmo de [Ashkiani et al. 2017]. Entretanto, como é necessário manter um *buffer* para cada *bin* na *shared memory*, a quantidade máxima de *bins* processáveis nessa abordagem torna-se limitada.

(iii) ***buffer+global:*** Embora utilize a *shared memory* para organizar as escritas na memória global, a versão *buffer-per-bin* pode se tornar ineficiente quando muitos elementos pertencem a um mesmo *bin*, pois todas as *threads* competem para inserir dados em um único *buffer*, causando contenção. Para contornar esse problema, a terceira abordagem, denominada *buffer+global*, detecta situações em que há excesso de escritas em um mesmo *buffer* e, nesse caso, as *threads* passam a gravar diretamente na memória global, sem utilizar o *buffer*. Ainda assim, permanece a restrição de manter um *buffer* por *bin* na *shared memory*, o que limita a quantidade máxima de *bins* que pode ser processada.

(iv) ***buffer-persistent:*** Por fim, a quarta implementação, denominada *buffer-persistent*, busca remover a limitação quanto ao número máximo de *bins*, assim como a versão *large*, mantendo, no entanto, a utilização de *buffers*. Para isso, emprega um *loop* que processa iterativamente N *bins* por vez, sendo N o número máximo de *buffers* que é possível alocar na *shared memory*. Essa abordagem permite processar um número arbitrário de *bins*. No entanto, como exige múltiplas leituras dos dados de entrada, o desempenho pode ser comprometido em cenários com grandes volumes de elementos e elevado número de *bins*.

Como as abordagens *buffer-per-bin* e *buffer+global* mantêm *buffers* na *shared memory*, a quantidade máxima de *bins* que pode ser processada nessa configuração é 361. Para valores superiores, podem ser utilizadas as abordagens *large* e *buffer-persistent*, capazes de processar até 12.288 *bins*, valor que corresponde ao número máximo de inteiros que pode ser armazenado em 48 KiB. As estratégias *large* e *buffer-per-bin* foram baseadas no trabalho de Cordeiro e Zola [Cordeiro and Nunan Zola 2025]. A implementação *buffer-per-bin*, detalhada no Algoritmo 1, é relevante por servir de base para outras abordagens baseadas em *buffer*. Sendo assim, optou-se por apresentar apenas esta versão em forma de algoritmo, uma vez que as demais derivam diretamente dela.

Algoritmo 1 Pseudocódigo do *buffer-per-bin:*

```
Require: Input, Output, BUFFER_SIZE = 32
 1: while existem elementos a processar do    // threads fazem em paralelo:
 2:     val ← próximo elemento
 3:     bin ← classificar(val)
 4:     toBeInserted ← true
 5:     while toBeInserted do
 6:         if bufSize[bin] < BUFFER_SIZE then
 7:             inserir val no buffer
 8:             toBeInserted ← false
 9:             if bufSize[bin] == BUFFER_SIZE − 1 then
10:                 toFlush ← true
11:             end if
12:         end if
13:         sincronizar threads
14:         warpFlush ← __ballot_sync(0xffffffff, toFlush)
15:         if warpFlush then    // threads do warp fazem em paralelo
16:             calcular posição de escrita em Output
17:             escrever elementos do buffer em Output (coalescido)
18:             zerar tamanho do buffer
19:         end if
20:     end while
21:     sincronizar threads
22: end while
23: for all buffers, cada warp do
24:     calcular posição de escrita final em Output
25:     escrever elementos restantes do buffer em Output (coalescido)
26:     zerar tamanho do buffer
27: end for
```

Inicialmente, cada *thread* classifica um elemento do conjunto de entrada e define a variável $toBeInserted = true$ (linhas 2 a 3), indicando que o elemento ainda precisa ser inserido no *buffer* local. Nas linhas 6 a 12, cada *thread* tenta inserir seu elemento no *buffer*. Se o *buffer* não estiver cheio ($tamanho < 32$), a *thread* insere o elemento e altera $toBeInserted = false$. Caso o *buffer* atinja 31 elementos, a *flag* $toFlush = true$ é acionada para sinalizar que o *buffer* está pronto para ser descarregado na memória

global. Todas as *threads* do *warp* executam uma votação coletiva (`__ballot_sync`) para determinar se algum *warp* precisa realizar o *flush* do *buffer*. Se algum *warp* sinalizar a necessidade de *flush*, todas as *threads* desse *warp* colaboram para copiar os elementos do *buffer* para a memória de saída de forma coalescida (linhas 15 a 19). Finalmente, nas linhas 23 a 27, após o término do *loop* principal, cada *warp* realiza um *flush* final, garantindo que todos os elementos restantes no *buffer* sejam corretamente escritos no vetor de saída na memória global.

Sendo assim, soluções que utilizam *buffer* seguem o modelo de programação *warp-centric*, no qual a tarefa de realizar escritas na memória global é atribuída a cada *warp*. Cada *warp* utiliza comunicações *intra-warp* para evitar a divergência de controle (*branch divergence*) e reduzir a latência de memória. Esse modelo tem demonstrado potencial para acelerar diversos algoritmos, como evidenciado em [Meyer et al. 2021] [Cordeiro and Nunan Zola 2023] [Ferraz et al. 2024].

Todas as funções CUDA (*CUDA kernels*) foram implementadas utilizando o modelo de programação de *threads persistentes* [Gupta et al. 2012]. Esse modelo apresenta ganhos de desempenho em diversos tipos de algoritmos em GPU, conforme descrito em diferentes estudos [Nunan Zola and De Bona 2012] [Cordeiro and Nunan Zola 2024]. Neste modelo, cada *thread* permanece ativa ao longo de toda a execução do *kernel*, até que não haja mais trabalho a ser processado. A principal vantagem da utilização de *threads persistentes* está na redução do número de blocos lançados na GPU e na manutenção desses blocos sempre ativos, o que permite preservar o estado de variáveis e reduzir a necessidade de comunicação via memória global. Dessa forma, evita-se o tempo adicional associado à recarga de dados em cada troca de contexto entre blocos de *threads*.

Será realizada uma análise comparativa entre essas quatro estratégias, avaliando o desempenho de cada abordagem para diferentes quantidades de elementos, números de *bins* e distribuições de dados. A partir dessa investigação, identificam-se as técnicas mais adequadas para cada cenário, que servirão de base para a construção de um algoritmo adaptativo, denominado *mrBins* (*multiSplit bins*), capaz de selecionar dinamicamente a estratégia mais eficiente conforme as características da entrada.

5. Metodologia dos Experimentos

Para avaliar a eficiência das estratégias desenvolvidas, foram realizados experimentos comparando-as com o *GPU multisplit* [Ashkiani et al. 2017]. Os conjuntos de dados foram gerados com tamanhos de 1, 8, 16, 32 e 64 milhões de elementos do tipo *unsigned int*, considerando distribuições uniforme, normal (gaussiana) e exponencial, descritas brevemente a seguir:

- **Distribuição uniforme:** todos os valores em um intervalo possuem a mesma probabilidade de ocorrência.
- **Distribuição normal:** os valores se concentram em torno de uma média. Neste trabalho, a média foi definida como "número total de elementos / 2" e o desvio padrão como "número total de elementos / 6".
- **Distribuição exponencial:** representa fenômenos em que a probabilidade decai rapidamente, de modo que valores pequenos são mais prováveis e valores grandes menos prováveis. O parâmetro lambda foi fixado em 0.01.

A quantidade de *bins* utilizada na execução de cada conjunto variou de 40 a 12.288. A função de categorização empregada nos experimentos atribui cada elemento

a um *bin* de acordo com a fórmula $f(x) = \lfloor x / \max \rfloor$, onde $\max$ representa o maior valor no conjunto de dados de entrada. Os testes foram repetidos 30 vezes, reportando a vazão média de elementos processados por segundo. Também foram calculados intervalos de confiança de 95%, não sendo observadas variações superiores a 0,5% em relação à média. Os experimentos foram conduzidos em um processador *Intel Xeon Silver* 4314 @ 2.40GHz, utilizando a versão 12.4 do CUDA e sistema operacional Linux Ubuntu 20.04.3 LTS. A GPU utilizada foi a NVIDIA RTX A4500, cuja arquitetura *Ampere* disponibiliza 96 KiB de *shared memory* por SM. Nos experimentos realizados, foi utilizado um máximo de 48 KiB de *shared memory* por bloco de *threads*, garantindo que múltiplos blocos pudessem residir simultaneamente em cada SM, o que melhora a ocupação. A partir dos resultados dos experimentos, será possível identificar quais técnicas são mais adequadas para cada cenário, possibilitando a construção do algoritmo adaptativo *mrBins*. Por fim, será realizada uma análise geral comparando o *mrBins* com o *GPU multisplit*, a fim de validar a abordagem desenvolvida.

6. Resultados e Discussões

N	Versão	Quantidade de bins						
		40	64	100	128	200	256	361
1 Milhão	GPU Multisplit	-	**22.28**	-	**18.84**	-	13.99	-
	large	13.86	12.98	12.31	12.11	11.70	11.46	11.27
	buffer-per-bin	11.65	11.66	16.17	16.15	**18.33**	**18.57**	**19.01**
	buffer+global	**17.37**	17.17	**17.02**	17.09	17.17	17.09	17.52
	buffer-persistent	16.37	16.33	16.32	16.30	16.26	16.43	17.13
	Speedup	-	0.77	-	0.91	-	1.33	-
8 Milhões	GPU Multisplit	-	**36.73**	-	27.20	-	18.65	-
	large	23.22	20.67	19.22	18.64	17.66	13.63	10.17
	buffer-per-bin	27.31	26.94	28.66	28.70	29.05	28.99	29.00
	buffer+global	**35.41**	34.32	**33.65**	**33.01**	**32.13**	**31.92**	**32.62**
	buffer-persistent	31.54	31.30	30.30	29.66	29.26	29.19	29.14
	Speedup	-	0.93	-	1.21	-	1.71	-
16 Milhões	GPU Multisplit	-	34.04	-	27.98	-	18.98	-
	large	13.82	11.82	10.88	10.56	10.17	10.03	8.83
	buffer-per-bin	31.17	30.65	31.21	31.10	30.95	30.87	30.83
	buffer+global	**38.76**	**37.46**	**36.66**	**35.02**	**34.05**	**34.06**	**34.58**
	buffer-persistent	34.16	33.73	32.56	31.68	31.25	31.18	31.10
	Speedup	-	1.10	-	1.25	-	1.79	-
32 Milhões	GPU Multisplit	-	34.75	-	27.97	-	17.96	-
	large	12.38	10.31	9.48	9.14	8.78	8.64	8.21
	buffer-per-bin	33.89	33.21	32.94	32.68	32.15	32.08	32.10
	buffer+global	**40.69**	**39.46**	**38.14**	**36.44**	**35.35**	**35.26**	**35.42**
	buffer-persistent	35.75	35.37	33.86	33.02	32.50	32.38	32.39
	Speedup	-	1.14	-	1.30	-	1.96	-
64 Milhões	GPU Multisplit	-	35.08	-	27.20	-	16.10	-
	large	11.91	9.89	8.85	8.52	8.19	8.05	7.83
	buffer-per-bin	35.44	34.72	33.86	33.56	32.75	32.73	32.73
	buffer+global	**41.85**	**40.53**	**38.84**	**37.12**	**35.82**	**35.80**	**35.90**
	buffer-persistent	36.63	36.21	34.66	33.70	33.14	33.09	33.05
	Speedup	-	1.16	-	1.36	-	2.22	-

Vazão do Multisplit (Bilhões de elementos processados/segundo)

Tabela esquerda: pequenas quantidades de bins — **Tabela direita: grandes quantidades de bins**

N	Versão	Quantidade de bins							
		512	1024	2048	3000	4096	5000	8192	12288
1 Milhão	GPU Multisplit	-	-	-	-	-	-	-	-
	large	11.13	10.49	**10.17**	**9.86**	**9.79**	**9.56**	**8.87**	**7.61**
	buffer-per-bin	-	-	-	-	-	-	-	-
	buffer+global	-	-	-	-	-	-	-	-
	buffer-persistent	**14.35**	**11.83**	8.19	6.76	5.11	4.81	3.34	2.29
8 Milhões	GPU Multisplit	-	-	-	-	-	-	-	-
	large	8.09	6.59	5.81	5.78	5.70	5.72	**5.62**	**5.52**
	buffer-per-bin	-	-	-	-	-	-	-	-
	buffer+global	-	-	-	-	-	-	-	-
	buffer-persistent	**21.45**	**17.27**	**11.38**	**9.86**	**7.11**	**6.83**	4.31	2.92
16 Milhões	GPU Multisplit	-	-	-	-	-	-	-	-
	large	7.29	5.80	5.30	5.14	4.97	5.00	**4.99**	**4.93**
	buffer-per-bin	-	-	-	-	-	-	-	-
	buffer+global	-	-	-	-	-	-	-	-
	buffer-persistent	**22.47**	**17.81**	**11.37**	**9.46**	**6.98**	**6.85**	4.45	3.00
32 Milhões	GPU Multisplit	-	-	-	-	-	-	-	-
	large	6.96	5.58	5.06	4.92	4.82	4.77	**4.67**	**4.68**
	buffer-per-bin	-	-	-	-	-	-	-	-
	buffer+global	-	-	-	-	-	-	-	-
	buffer-persistent	**23.10**	**18.06**	**11.43**	**9.44**	**7.47**	**6.69**	4.30	3.02
64 Milhões	GPU Multisplit	-	-	-	-	-	-	-	-
	large	6.75	5.47	4.87	4.71	4.63	4.56	**4.49**	**4.44**
	buffer-per-bin	-	-	-	-	-	-	-	-
	buffer+global	-	-	-	-	-	-	-	-
	buffer-persistent	**23.43**	**18.18**	**11.45**	**8.54**	**6.79**	**6.00**	3.91	2.71

Tabela 1. Resultado dos experimentos com *datasets* gerados a partir de uma distribuição uniforme. Os valores correspondem à vazão de bilhões de elementos processados por segundo. Resultados em rosa indicam desempenho inferior ao *GPU multisplit*, enquanto resultados em verde indicam desempenho superior. O *speedup* em relação ao *GPU multisplit* é calculado com base na abordagem que apresentou o melhor resultado. Valores anotados com "–" representam limitações do algoritmo *GPU multisplit*.

Os resultados dos experimentos com a distribuição uniforme estão detalhados na Tabela 1. Primeiramente, observa-se que as implementações desenvolvidas apresentam bom desempenho com o aumento da quantidade de elementos e de *bins*. Com exceção da versão *large*, todas as versões superaram o *GPU multisplit* para mais de 16 milhões de elementos e acima de 128 *bins*, alcançando aceleração de até 2,22 vezes em relação ao estado

Vazão do Multisplit (Bilhões de elementos processados/segundo)

Tabela à esquerda: Dados gerados seguindo uma distribuição normal

N	Versão	Quantidade de bins						
		40	64	100	128	200	256	361
1 Milhão	GPU Multisplit	-	**23.46**	-	**20.13**	-	14.84	-
	large	16.16	14.49	13.36	13.24	12.70	12.44	11.37
	buffer-per-bin	11.56	11.50	15.62	15.77	**17.77**	**18.02**	**17.74**
	buffer+global	**17.23**	17.40	**17.14**	17.15	16.99	17.12	16.66
	buffer-persistent	15.33	15.87	16.25	16.21	16.08	15.97	15.96
	Speedup	-	0.74	-	0.85	-	1.21	-
8 Milhões	GPU Multisplit	-	**37.20**	-	30.87	-	21.26	-
	large	30.87	25.82	23.25	22.09	21.12	20.48	14.79
	buffer-per-bin	30.05	29.28	31.68	31.03	31.42	30.91	30.40
	buffer+global	**37.47**	36.97	**36.33**	**35.82**	**34.42**	**33.46**	**32.78**
	buffer-persistent	30.99	33.07	33.19	32.60	31.64	30.86	30.58
	Speedup	-	0.99	-	1.16	-	1.57	-
16 Milhões	GPU Multisplit	-	37.77	-	30.69	-	21.02	-
	large	20.29	15.65	13.60	12.87	12.04	11.76	11.46
	buffer-per-bin	34.33	33.26	34.39	33.56	33.30	32.75	32.23
	buffer+global	**40.66**	**39.64**	**39.01**	**38.10**	**36.08**	**35.20**	**34.52**
	buffer-persistent	33.26	35.94	35.55	34.94	33.60	33.01	32.48
	Speedup	-	1.05	-	1.24	-	1.67	-
32 Milhões	GPU Multisplit	-	38.07	-	30.70	-	20.82	-
	large	18.54	14.15	11.98	11.28	10.53	10.23	9.95
	buffer-per-bin	37.10	35.80	36.05	35.24	34.58	33.93	33.41
	buffer+global	**42.51**	**41.59**	**40.52**	**39.76**	**37.48**	**36.35**	**35.48**
	buffer-persistent	34.52	37.26	37.04	36.39	34.85	34.19	33.69
	Speedup	-	1.09	-	1.30	-	1.75	-
64 Milhões	GPU Multisplit	-	37.78	-	29.99	-	19.19	-
	large	17.53	13.28	11.28	10.59	9.73	9.45	9.19
	buffer-per-bin	38.56	37.24	36.87	36.04	35.13	34.46	33.96
	buffer+global	**43.53**	**42.63**	**41.62**	**40.47**	**37.81**	**36.99**	**36.02**
	buffer-persistent	35.02	37.97	37.73	37.03	35.46	34.79	34.31
	Speedup	-	1.13	-	1.35	-	1.93	-

Tabela à direita: Dados gerados seguindo uma distribuição exponencial

N	Versão	Quantidade de bins						
		40	64	100	128	200	256	361
1 Milhão	GPU Multisplit	-	**22.25**	-	**19.32**	-	14.52	-
	large	**18.44**	18.43	16.80	15.74	14.72	13.68	12.98
	buffer-per-bin	11.15	11.36	14.97	15.14	**16.65**	**16.84**	**16.91**
	buffer+global	17.44	17.45	**17.19**	17.20	16.94	16.59	16.20
	buffer-persistent	11.63	13.00	13.80	14.58	15.00	14.96	15.34
	Speedup	-	0.83	-	0.89	-	1.16	-
8 Milhões	GPU Multisplit	-	37.36	-	31.29	-	22.22	-
	large	**39.18**	**38.82**	38.00	37.12	31.50	29.28	26.48
	buffer-per-bin	26.71	28.84	28.00	29.37	28.92	29.99	30.62
	buffer+global	37.29	37.32	**37.38**	**37.22**	**37.28**	**36.80**	**36.27**
	buffer-persistent	17.73	21.61	25.14	26.76	29.61	30.67	31.17
	Speedup	-	1.04	-	1.19	-	1.66	-
16 Milhões	GPU Multisplit	-	38.49	-	31.65	-	22.24	-
	large	**42.15**	**41.28**	32.54	28.20	21.41	18.77	16.28
	buffer-per-bin	29.87	32.75	30.08	31.74	30.76	31.92	32.69
	buffer+global	40.30	40.31	**40.28**	**40.22**	**40.14**	**40.10**	**39.52**
	buffer-persistent	18.67	22.85	26.70	28.49	31.75	32.97	33.56
	Speedup	-	1.07	-	1.27	-	1.80	-
32 Milhões	GPU Multisplit	-	39.20	-	32.01	-	22.42	-
	large	**44.48**	41.72	29.35	25.54	19.02	16.72	14.51
	buffer-per-bin	32.19	35.54	31.69	33.46	32.02	33.38	34.21
	buffer+global	42.41	**42.13**	**42.28**	**42.43**	**42.36**	**42.05**	**41.51**
	buffer-persistent	18.92	23.34	27.45	29.37	32.94	34.24	35.04
	Speedup	-	1.07	-	1.33	-	1.88	-
64 Milhões	GPU Multisplit	-	39.05	-	31.65	-	21.54	-
	large	**45.49**	37.95	27.81	24.19	17.99	15.76	13.64
	buffer-per-bin	33.26	36.88	32.34	34.20	32.58	33.96	34.84
	buffer+global	43.39	**43.32**	**43.44**	**43.43**	**43.45**	**43.10**	**42.41**
	buffer-persistent	18.97	23.50	27.69	29.67	33.40	34.79	35.64
	Speedup	-	1.11	-	1.37	-	2.00	-

Tabela 2. Resultado dos experimentos com datasets gerados a partir das distribuições normal (à esquerda) e exponencial (à direita). Assim como na Tabela 1, os valores correspondem à vazão de bilhões de elementos processados por segundo. Resultados em rosa indicam desempenho inferior ao *GPU multisplit*, enquanto os em verde indicam desempenho superior. O *speedup* em relação ao *GPU multisplit* é calculado com base na melhor abordagem em cada caso. Valores anotados com "–" representam limitações do algoritmo *GPU multisplit*.

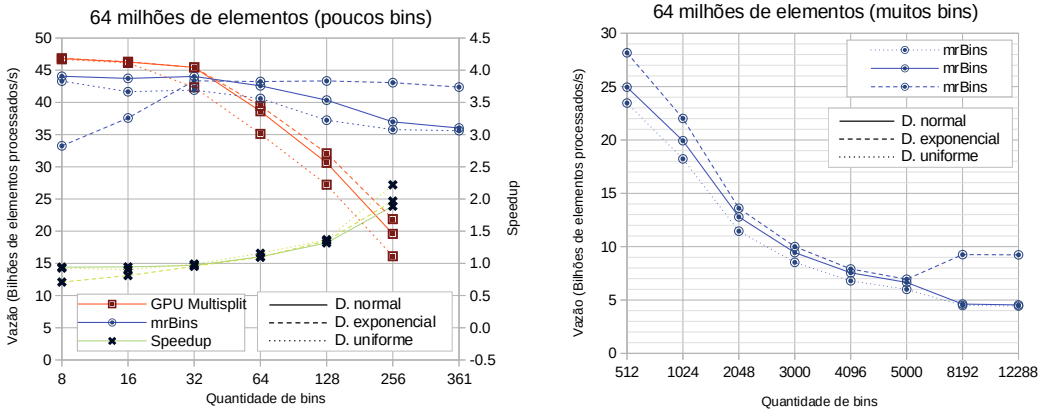

Figura 1. Resultado da comparação entre o algoritmo *mrBins* e o *GPU Multisplit*, variando a quantidade de *bins* com datasets de 64 milhões de elementos. À esquerda, a quantidade de *bins* variou de 8 a 361, e à direita, de 512 a 12.288. A linha contínua representa os resultados para a distribuição normal, a tracejada para a distribuição exponencial e a pontilhada para a distribuição uniforme. A linha verde indica o *speedup* do *mrBins* em relação ao *GPU Multisplit*.

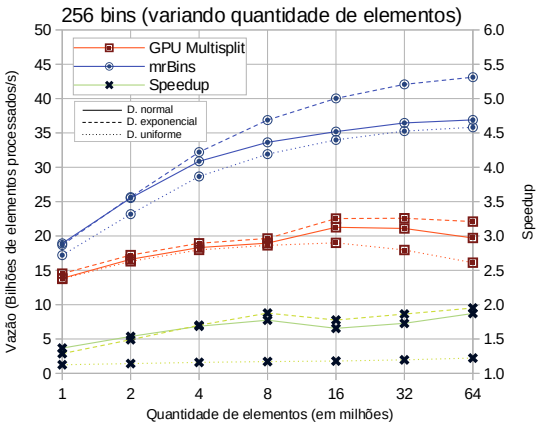

Figura 2. Resultado da comparação entre o algoritmo *mrBins* e o *GPU multisplit*, variando a quantidade de elementos. A quantidade de *bins* foi fixada em 256. A linha contínua representa os resultados para a distribuição normal, a tracejada para a distribuição exponencial e a pontilhada para a distribuição uniforme. A linha verde indica o *speedup* do *mrBins* em relação ao *GPU Multisplit*.

da arte. Entre as versões baseadas em *buffer*, a que mais se destacou foi a *buffer+global*, devido à sua estratégia de evitar contenções nos *buffers*. Para mais de 8 milhões de elementos, a versão *buffer+global* superou as demais implementações propostas em todos os cenários.

No cenário com 1 milhão de elementos, a versão *buffer-per-bin* apresentou melhor desempenho, devido à sua abordagem mais simples, que facilita o tratamento de *datasets* pequenos. Ainda assim, a diferença em relação à versão *buffer+global* não foi significativa. Analisando o cenário com grandes quantidades de *bins* (à direita da Tabela 1), observa-se que, em geral, a versão *buffer-persistent* apresentou melhor desempenho para grandes volumes de elementos e quantidade média de *bins*, superando a versão *large* em *datasets* com mais de 8 milhões de elementos e até 5 mil *bins*. Esse resultado se deve ao fato de que, embora as escritas sejam realizadas de forma coalescida, quando a quantidade de *bins* cresce muito, essa versão precisa ler o vetor de dados diversas vezes.

Para a distribuição normal, apresentada à esquerda na Tabela 2, os resultados permaneceram semelhantes, com a versão *buffer+global* apresentando desempenho superior às demais versões em todas as quantidades de *bins* para mais de 8 milhões de elementos. Já para a distribuição exponencial, mostrada à direita na Tabela 2, que representa um desafio maior para abordagens baseadas em *buffer* devido à concentração de muitos elementos em poucos *bins*, a versão *large* se destacou em cenários com baixa quantidade de *bins*. Isso ocorre porque, como muitos *bins* estão próximos na memória, as escritas coalescidas acontecem automaticamente, sem a necessidade de utilizar estratégias adicionais, que, nesse caso, podem até prejudicar o desempenho, como ocorreu com as versões baseadas em *buffer*. Ainda assim, a versão *buffer+global* apresentou desempenho próximo ao da *large* nos melhores casos e manteve resultados mais consistentes quando a *large* apresentou quedas de eficiência. Em relação ao *GPU multisplit*, os resultados da Tabela 2 são muito semelhantes aos da Tabela 1, com a *buffer+global* superando-o em todos os experimentos com mais de 64 *bins* e 16 milhões de elementos.

Dessa forma, o algoritmo *mrBins* pode ser implementado da seguinte forma: para

quantidades de *bins* até 361, utiliza-se a versão *buffer+global*, que se mostrou mais eficiente em relação às outras versões na maioria dos cenários. Para quantidades de *bins* até 5.000, é utilizado a versão *buffer-persistent* e para tamanhos maiores, utiliza-se a versão *large*. Apesar de a versão *buffer-per-bin* ter servido como base de desenvolvimento, ela não é utilizada no *mrBins*. As Figuras 1 e 2 apresentam a comparação entre o *mrBins* e o *GPU Multisplit*. Os resultados indicam, conforme mostrado nas tabelas, que o *mrBins* se destaca quando a quantidade de *bins* é superior a 64 e o número de elementos é maior que 1 milhão. Entretanto, estratégias para otimizar o algoritmo para cenários com menos de 64 *bins* ainda são necessárias, sendo consideradas como trabalho futuro.

7. Conclusões

Este trabalho teve como objetivo propor um algoritmo eficiente para multiparticionamento em GPU. Foram apresentadas e analisadas quatro soluções, e, a partir dessa análise, foi proposto o *mrBins*, algoritmo capaz de escolher, em tempo de execução, a melhor estratégia para lidar com os dados de entrada, dependendo da quantidade de *bins*. As versões, e, subsequentemente, o *mrBins*, foram comparadas com o algoritmo estado da arte, *GPU multisplit*.

Experimentos realizados demonstram que o *mrBins* possui potencial para superar o desempenho do *GPU multisplit* em diversos cenários, principalmente quando a quantidade de *bins* e o tamanho do *dataset* são elevados, alcançando aceleração de até 2,2 vezes. A escolha adaptativa das versões do algoritmo (*buffer+global*, *buffer-persistent* e *large*) permite otimizar o desempenho de acordo com o número de *bins*, garantindo eficiência consistente independentemente da distribuição dos dados, seja uniforme, normal ou exponencial. Como trabalhos futuros, serão estudadas novas otimizações para cenários com menos de 64 *bins*, bem como a utilização da maior capacidade de trabalho em *shared memory*, como é o caso de arquiteturas mais recentes, como a *Ampere*.

O algoritmo desenvolvido neste artigo será disponibilizado em `https://github.com/MichelBC/MrBins`.

Agradecimentos

Este trabalho foi parcialmente suportado pelo Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq), processo 407644/2021-0, bem como pela Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES) - Programa de Excelência Acadêmica (PROEX).

Referências

Alcantara, D. A., Sharf, A., Abbasinejad, F., Sengupta, S., Mitzenmacher, M., Owens, J. D., and Amenta, N. (2009). Real-time parallel hashing on the GPU. In *ACM SIGGRAPH asia 2009 papers*, pages 1–9.

Ashkiani, S., Davidson, A., Meyer, U., and Owens, J. D. (2016). GPU multisplit. *SIGPLAN Not.*, 51(8).

Ashkiani, S., Davidson, A., Meyer, U., and Owens, J. D. (2017). GPU Multisplit: an extended study of a parallel algorithm. *ACM Transactions on Parallel Computing*.

Choi, B., Komuravelli, R., Lu, V., Sung, H., Bocchino Jr, R. L., Adve, S. V., and Hart, J. C. (2010). Parallel SAH kD-tree construction. In *High performance graphics*, pages 77–86. Citeseer.

Cordeiro, M. B., Blanco, R. M., and Nunan Zola, W. M. (2025a). Algoritmo paralelo e distribuído para ordenação chave-valor. In *Anais da XX Escola Regional de Banco de Dados*, pages 133–136, Porto Alegre, RS, Brasil. SBC.

Cordeiro, M. B., Bluhm, G., and Nunan Zola, W. M. (2025b). Multiparticionamento flexível em processadores multicore. In *Anais da XXV Escola Regional de Alto Desempenho da Região Sul*, pages 169–170, Porto Alegre, RS, Brasil. SBC.

Cordeiro, M. B. and Nunan Zola, W. M. (2023). KNN paralelo em gpu para grandes volumes de dados com agregação de consultas. In *Simpósio em Sistemas Computacionais de Alto Desempenho (SSCAD)*, pages 253–264. SBC.

Cordeiro, M. B. and Nunan Zola, W. M. (2024). Parallel k-means on GPU using warp-centric strategies. In *2024 IEEE 30th International Conference on Parallel and Distributed Systems (ICPADS)*, pages 720–727.

Cordeiro, M. B. and Nunan Zola, W. M. (2025). Multiparticionamento de dados em gpu. In *Anais da XXV Escola Regional de Alto Desempenho da Região Sul*, pages 165–166, Foz do Iguaçu, PR, Brasil. SBC.

Ferraz, S., Dias, V., Teixeira, C. H., Parthasarathy, S., Teodoro, G., and Meira, W. (2024). DuMato: An efficient warp-centric subgraph enumeration system for GPU. *Journal of Parallel and Distributed Computing*, 191:104903.

Gupta, K., Stuart, J. A., and Owens, J. D. (2012). A study of persistent threads style GPU programming for GPGPU workloads. In *IEEE 2012 Innovative Parallel Computing (InPar)*. IEEE.

Jünger, D., Kobus, R., Müller, A., Hundt, C., Xu, K., Liu, W., and Schmidt, B. (2020). WarpCore: A library for fast hash tables on GPUs. In *2020 IEEE 27th International Conference on High Performance Computing, Data, and Analytics (HiPC)*.

Lessley, B. and Childs, H. (2020). Data-parallel hashing techniques for GPU architectures. *IEEE Transactions on Parallel and Distributed Systems*, 31(1):237–250.

Merrill, D. G. and Grimshaw, A. S. (2010). Revisiting sorting for GPGPU stream architectures. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, pages 545–546.

Meyer, B., Pozo, A., and Nunan Zola, W. M. (2021). Warp-centric k-nearest neighbor graphs construction on GPU. In *50th International Conference on Parallel Processing Workshop*, ICPP Workshops '21, New York, NY, USA. Association for Computing Machinery.

Nunan Zola, W. M. and De Bona, L. C. E. (2012). Parallel speculative encryption of multiple AES contexts on GPUs. In *2012 Innovative Parallel Computing (InPar)*, pages 1–9. IEEE.

Reinders, J. (2007). *Intel threading building blocks: outfitting C++ for multi-core processor parallelism*. "O'Reilly Media, Inc.".

Stehle, E. and Jacobsen, H.-A. (2017). A memory bandwidth-efficient hybrid radix sort on GPUs. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, New York, NY, USA. Association for Computing Machinery.