Redução Paralela Otimizada para Segmentos Regulares e Irregulares em GPU

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

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

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

Abstract. *Reduction is an operation that combines all the elements of a collection by applying a binary operation, such as sum, maximum, or minimum, to all the elements to obtain a single resulting value. This work aims to investigate implementation strategies for segmented reduction on GPUs. Existing techniques for segmented reduction often show consistent performance but are relatively inefficient in absolute terms. These techniques are frequently optimized for specific workloads and, as a result, may exhibit degraded performance with certain input data, especially when segments have varying sizes. The algorithm presented in this paper employs three different strategies to handle segments of varying sizes and has the capability to select the best strategy at runtime to optimize the processing of each segment based on its size. The proposed algorithm delivers consistent performance, achieving up to 49.47 times speedup reducing variable-sized segments and up to 12.62 times acceleration working with regular-sized segments, compared to the segmented reduction algorithm from top GPU libraries. Additionally, this paper also explores strategies to accelerate non-segmented reduction, resulting in up to 1.77 times improvement compared to other parallel GPU implementations.*

Resumo. *Redução é uma operação que combina todos os elementos de uma coleção aplicando uma operação binária, como soma, máximo ou mínimo, a todos os elementos para obter um único valor resultante. Este trabalho tem como objetivo investigar estratégias de implementação para redução segmentada em GPUs. Técnicas existentes de redução segmentada costumam ter um desempenho consistente, mas são relativamente ineficientes quando aplicadas a segmentos irregulares. Essas técnicas são frequentemente otimizadas para cargas de trabalho específicas e, como resultado, podem apresentar desempenho degradado para certos conjuntos de dados, especialmente quando os segmentos têm tamanhos muito variados. O algoritmo apresentado neste artigo utiliza três estratégias diferentes para lidar com tamanhos variados de segmentos e tem a capacidade de escolher a melhor estratégia em tempo de execução para otimizar o processamento de cada segmento conforme seu tamanho. O algoritmo proposto oferece um desempenho consistente, alcançando uma aceleração de até 49.47 vezes para segmentos de tamanhos variados e até 12.62 vezes para segmentos de tamanhos regulares, em comparação aos algoritmos de redução segmentada das melhores bibliotecas paralelas em GPU. Além disso, este artigo também explora estratégias para acelerar a redução não segmentada, resultando em uma melhoria de até 1.77 vezes em comparação às outras implementações.*

1. Introdução

Redução é uma operação comum em programação paralela, que consiste em aplicar uma operação binária a todos os elementos de uma coleção de dados para produzir um único resultado. Esta técnica é amplamente utilizada em várias áreas da computação e frequentemente serve como um passo fundamental em muitos algoritmos, como Backpropagation e K-means [Che et al. 2009], KNN [Warashina et al. 2014] [Cordeiro and Zola 2023], Stream Compaction [Johnson et al. 2019] etc.

Uma operação de redução pode ser formalmente descrita da seguinte maneira: dado um conjunto X com n elementos, $X = \{x_0, x_1, ..., x_{n-1}\}$, a redução consiste em calcular $x_0 \otimes x_1 \otimes ... \otimes x_{n-1}$, onde $\otimes$ representa uma função de combinação que deve ser associativa e comutativa. Exemplos comuns de operadores $\otimes$ quando X é composto por números incluem $+$ (soma), $\times$ (multiplicação), $\wedge$ (e lógico), $\vee$ (ou lógico), $\oplus$ (ou exclusivo), $\cap$ (interseção), $\cup$ (união), max (máximo) e min (mínimo).

Uma variação da operação de redução é a redução segmentada, cujo objetivo é aplicar uma operação binária a S segmentos independentes de um conjunto de dados. Sendo assim, o resultado da redução é um vetor de tamanho S, com um valor para cada segmento do conjunto de dados. Esse tipo de operação é frequentemente utilizado em matrizes, onde se deseja realizar uma redução de forma independente em cada linha da matriz. Embora seja comum, os segmentos não precisam ser regulares, ou seja, não precisam ter o mesmo tamanho. Enquanto a redução não segmentada em GPUs é um problema de paralelismo amplamente estudado e bem consolidado na literatura [Harris 2007] [Luitjens 2014] [Jradi et al. 2018], a redução segmentada tem recebido menos atenção e frequentemente enfrenta desafios relacionados à eficiência [Larsen and Henriksen 2017]. Apesar de oferecer um desempenho consistente, as técnicas atuais de redução segmentada costumam ser otimizadas para cenários específicos e podem não escalar de maneira eficiente quando os segmentos têm tamanhos variados. Essa limitação pode resultar em uma degradação significativa no desempenho, especialmente em aplicações que lidam com conjuntos de dados em que os segmentos apresentam tamanhos muito diferentes.

Este trabalho tem como objetivo aprimorar os algoritmos de redução segmentada e investigar métodos para acelerar a redução não segmentada em GPUs. O algoritmo proposto utiliza três abordagens distintas otimizadas para diferentes tamanhos de segmentos e é projetado para escolher, em tempo de execução, a estratégia mais adequada para realizar a redução em cada segmento. Os algoritmos desenvolvidos são então comparados com os algoritmos das bibliotecas Thrust [Thrust 2024] e CUB [NVIDIA 2024], ambas amplamente utilizadas em aplicações de alto desempenho em GPUs.

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

Como mencionado anteriormente, a paralelização da operação de redução em GPUs é um problema amplamente estudado. Em seu trabalho, [Harris 2007] descreve técnicas

para lidar com grandes conjuntos de dados e demonstra, através de várias versões do algoritmo, como escolhas inadequadas no mapeamento do problema podem impactar negativamente o desempenho. São discutidos também desafios associados à programação em GPUs, como leitura não coalescida da memória, comunicação ineficiente e alta divergência de *warps*. Ao mitigar comandos condicionais, manter todas as *threads* ativas o máximo possível e promover a cooperação entre *threads* do mesmo *warp*, Harris conseguiu alcançar uma aceleração de até 30 vezes em relação à versão inicial do algoritmo.

A partir desse trabalho, [Luitjens 2014] demonstra como otimizar ainda mais o algoritmo de Harris utilizando instruções de *shuffle*, disponíveis a partir da arquitetura Kepler das GPUs da NVIDIA. Essas instruções permitem que *threads* dentro de um mesmo *warp* troquem dados diretamente entre registradores, eliminando a necessidade de recorrer às memórias de comunicação, como a memória compartilhada ou a memória global. Segundo o autor, essa abordagem é superior à de Harris, pois a comunicação por *shuffle* é mais eficiente do que o uso de memória compartilhada. No entanto, Luitjens não apresenta um estudo comparativo entre os dois métodos. Portanto, este artigo implementa ambos os algoritmos e realiza uma comparação detalhada entre eles.

Outro trabalho que se propôs a acelerar a redução em GPUs foi conduzido por [Jradi et al. 2018]. Neste estudo, os autores apresentaram uma estratégia que é suficientemente genérica para ser usada tanto com CUDA quanto com OpenCL e pode rodar em hardware dos dois principais fabricantes de GPUs com mudanças mínimas. O código implementado, além de mais simples, ofereceu um desempenho equivalente à melhor estratégia descrita por Luitjens.

Por último, o estudo conduzido por [Larsen and Henriksen 2017] apresenta uma técnica para redução segmentada em GPUs. Nesse trabalho, são discutidas estratégias para realizar a redução em conjuntos de dados com segmentos de tamanhos iguais, e é demonstrada uma implementação no compilador Futhark. A implementação é comparada com o algoritmo de redução segmentada da biblioteca CUB. Como resultado, embora o CUB se destaque para entradas com segmentos cujos tamanhos estão em um intervalo específico, ele apresenta um desempenho ruim para casos extremos com muitos segmentos pequenos ou poucos grandes segmentos.

3. Fundamentos Teóricos

A versão sequencial do algoritmo de redução consiste em percorrer o conjunto de dados de forma linear, aplicando o operador binário a cada elemento e armazenando o resultado em um acumulador. A versão paralela realiza a computação do operador binário de maneira concorrente em múltiplos elementos, utilizando várias unidades de execução. Isso é possível sempre que o operador binário for associativo e comutativo. Nesses casos, o problema pode ser dividido hierarquicamente em subproblemas que são resolvidos em paralelo, com os resultados parciais sendo combinados posteriormente para obter o resultado final. Se o operador binário for tanto associativo quanto comutativo, a ordem na qual os resultados parciais são combinados não afetará o resultado final. A paralelização da redução em GPU apresenta os seguintes desafios principais:

- **Granularização do problema:** Dividir o problema de forma eficaz para que a carga de trabalho seja bem distribuída entre as *threads*, evitando a criação de *threads* ociosas e maximizando o uso dos recursos da GPU.

- **Comunicação entre *threads*:** Minimizar a necessidade de comunicação entre *threads* para reduzir a latência associada ao acesso a memória ou a sincronização.
- **Leitura eficiente da memória:** Garantir que as *threads* acessem a memória de forma coalescida para evitar penalidades de desempenho associadas a leituras da memória global.
- **Divergência de *warp*:** Evitar a divergência de *warp*, que ocorre quando um desvio condicional causa bifurcações no fluxo de execução das *threads* dentro de um *warp*, resultando em diferentes caminhos de execução e degradando o desempenho devido à execução serializada de *threads* divergentes.

Esses desafios são intensificados na redução segmentada, principalmente quando os segmentos são irregulares. Tamanhos variáveis de segmentos dificultam o equilíbrio da carga de trabalho entre as *threads*, tornando mais complexo garantir que todas estejam igualmente ocupadas. O acesso não coalescido à memória pode ocorrer com mais frequência devido às *threads* estarem percorrendo segmentos de tamanhos diferentes e potencialmente dispersos na memória. A divergência de *warp* pode ser mais frequente na redução segmentada, já que segmentos diferentes podem levar a caminhos de execução distintos para *threads* dentro do mesmo *warp*, impactando negativamente o desempenho.

3.1. Biblioteca CUB

CUB [NVIDIA 2024] é uma biblioteca de algoritmos paralelos desenvolvida pela NVIDIA que é projetada para fornecer componentes de alto desempenho e de baixo nível para operações paralelas comuns, como reduções, *scans*, e ordenações, que são essenciais para o desenvolvimento de algoritmos paralelos eficientes. A biblioteca CUB é amplamente utilizada em aplicações paralelas em GPU.

Para reduções não segmentadas, o CUB utiliza uma abordagem de redução paralela tradicional. A ideia é dividir o problema entre os blocos de *threads*, onde cada bloco processa uma parte do trabalho. Cada *thread* dentro do bloco realiza uma parte do cálculo, e depois os resultados parciais são combinados para produzir o resultado final. Para reduções segmentadas, o CUB adota uma estratégia diferente. A biblioteca lança um bloco de *threads* para cada segmento individual, e cada bloco realiza a redução para o segmento correspondente. Isso significa que cada segmento é processado de forma independente pelos blocos, o que pode ser eficiente quando o número de segmentos está dentro de um intervalo ideal. No entanto, o desempenho do CUB pode deteriorar em alguns casos, como quando há muitos segmentos pequenos ou poucos segmentos grandes.

3.2. Biblioteca Thrust

Assim como o CUB, Thrust [Thrust 2024] é uma biblioteca de algoritmos paralelos para CUDA C++ desenvolvida pela NVIDIA. Ela oferece uma interface de programação de alto nível para operações paralelas em GPUs NVIDIA, facilitando a implementação de algoritmos paralelos. Embora a biblioteca Thrust inclua uma implementação para o algoritmo de redução, ela não oferece diretamente uma operação de redução segmentada. Porém, é possível realizar reduções segmentadas utilizando Thrust através da função de redução por chave. Para isso, os elementos pertencentes ao mesmo segmento devem ser inicializados com a mesma chave. A função `thrust::reduce_by_key` então aplica a redução a cada grupo de elementos com a mesma chave, produzindo resultados equivalentes aos de um algoritmo de redução segmentada. É importante ressaltar que, em-

bora essa abordagem funcione, ela pode ter uma latência maior em comparação com uma implementação dedicada de redução segmentada.

4. Estratégias de Implementação

Primeiramente, é importante discutir as estratégias para a implementação da redução não segmentada. Neste cenário, as *threads* devem percorrer o conjunto de entrada de maneira coalescida, aplicando a operação binária. Para otimizar o desempenho, o uso do modelo de *kernel* persistente [Gupta et al. 2012] [Zola and De Bona 2012] pode ser vantajoso, pois permite que cada *thread* permaneça ativa por mais tempo. Isso resulta em cada *thread* processando mais dados de entrada, reduzindo a necessidade de comunicação entre elas. Após o processamento completo do conjunto de dados, os resultados parciais de cada *thread* dentro de um bloco devem ser combinados, e, em seguida, os resultados agregados de todos os blocos devem ser consolidados em um único elemento final.

Assim, a fase inicial, na qual as *threads* percorrem o conjunto de entrada acumulando resultados parciais, é uma etapa comum em todas as implementações de redução em GPU. Porém, a etapa de combinação dos resultados parciais dentro de um bloco pode variar significativamente e influenciar o resultado final.Portanto, neste trabalho, são implementadas e comparadas as seguintes estratégias para esta segunda etapa da redução:

- **ui32_sh_mem:** Esta implementação, baseada no trabalho de [Harris 2007], realiza a redução final na memória compartilhada da GPU.
- **ui32_shfl:** Esta abordagem, fundamentada no trabalho de [Luitjens 2014], utiliza operações de *shuffle* para combinar os resultados parciais.
- **ui32_atomic_op:** Nesta implementação, cada *thread* do bloco executa uma operação atômica em uma variável armazenada na memória compartilhada para agregar os resultados.
- **ui32x4:** Esta implementação também utiliza operações atômicas para obter o resultado final do bloco, mas realiza leituras de dados de forma vetorizada, processando 4 elementos de cada vez.

4.1. Redução Segmentada

Na redução segmentada, como os segmentos podem ter quantidades variáveis de elementos, não é possível utilizar um *kernel* único para processar cada segmento de maneira eficiente. Isso ocorre porque muitas *threads* ficarão ociosas se a quantidade de elementos em um segmento não for suficiente para ocupar toda a capacidade da GPU. Além disso, usar um bloco por segmento pode não ser eficiente. Se o número de segmentos for menor do que o número de blocos necessário para preencher a GPU, o algoritmo subutilizará os recursos de hardware. Por outro lado, se o tamanho dos segmentos for menor do que o número de *threads* por bloco, algumas *threads* ficarão ociosas, resultando em um desempenho inferior ao máximo possível. Outra abordagem seria atribuir cada segmento a um *warp*. Essa abordagem *warp centric* [Meyer et al. 2021] [Ferraz et al. 2024] evitaria a divergência de *warp*, pois todas as *threads* estariam trabalhando dentro do mesmo segmento. No entanto, essa estratégia também pode ser ineficiente se o número de segmentos for menor do que o número de *warps* disponíveis na GPU. Portanto, é evidente que é necessário um *kernel* adaptativo que escolha a granularidade mais adequada com base no tamanho dos segmentos.

Sendo assim, este trabalho propõe o algoritmo `bestReduce`, descrito no Algoritmo 1. A principal estratégia do `bestReduce` é escolher a melhor granularidade para realizar a redução, adaptando-se dinamicamente durante a execução do *kernel*. Foram implementados três níveis de granularidade:

Algoritmo 1: bestReduce

```
__global__
void bestReduce( DATA_TYPE* In, //vetor de entrada
  __shared__ int blockReduceNeeded = 0;
   uint* segStart, //inicio de cada segmento no vetor In
   int nSeg,  //quantidade de segmentos
   List* bigSeg, //segmentos a serem reduzidos pelo kernelReduce
   DATA_TYPE* Out ){ //resultado da reducao para cada segmento
  int nWarps = (blockDim*gridDim)/WARP_SIZE; //total de warp
  int wid = (threadIdx+blockDim*blockIdx)/WARP_SIZE; //warp id
  for(int seg = wid; seg < nSeg; seg+=nWarps){
    int segTam = segStart[seg+1] - segStart[seg];
    if(segTam < BEST_WARP_REDUCE)
      warpReduce();
    else if(segTam < BEST_BLOCK_REDUCE)
      blockReduceNeeded = 1;
    else
      addToList(seg, bigSeg);
    if(blockReduceNeeded){
      __syncthreads();
      blockReduce();
      blockReduceNeeded = 0;
    }
  }
}
%\vspace{-2mm}
```

- **warpReduce:** Para segmentos pequenos, cada *thread* do percorre o segmento aplicando a redução. Após o processamento, as *threads* combinam os resultados parciais usando instruções de *shuffle*, e uma *thread* é responsável por salvar o resultado final no vetor de saída.
- **blockReduce:** Para segmentos de tamanho médio, cada *thread* do bloco processa seus elementos e realiza a redução. Ao final, os resultados parciais são combinados usando operações atômicas, salvando o resultado no vetor de saída.
- **kernelReduce:** Para vetores grandes, o algoritmo armazena os segmentos em uma lista. Em seguida, um *kernel* de redução é lançado para processar cada segmento da lista. O `kernelReduce` é baseado na implementação `ui32x4` para redução não segmentada. Como o segmento é grande, ele pode ser tratado como um conjunto independente sem que uma quantidade considerável de *threads* fiquem ociosas. A implementação `ui32x4` foi escolhida por oferecer melhor desempenho para redução não segmentada, conforme Seção 6.

Os parâmetros `BEST_WARP_REDUCE` e `BEST_BLOCK_REDUCE` definem a granularidade a ser utilizada para cada tamanho de segmento. Esses parâmetros dependem

da arquitetura da GPU, como a quantidade de *cores* e multiprocessadores, e, portanto, é necessário que sejam definidos antes da compilação. Na Seção 6 são realizados testes para identificar os melhores parâmetros para o hardware utilizado nos experimentos.

5. Metodologia

Os experimentos serão divididos em duas etapas. A primeira etapa focará na redução não segmentada. Os algoritmos comparados foram apresentados na Seção 4. Além disso, serão avaliados também os algoritmos de redução das bibliotecas CUB [NVIDIA 2024] e Thrust [Thrust 2024]. Nesta etapa, serão gerados conjuntos de dados com tamanhos que variam de 128 mil a 64 milhões de elementos. Na segunda etapa, será avaliado o desempenho dos algoritmos de redução segmentada. Os algoritmos analisados são: o bestReduce, proposto na Seção 4, o `cub::DeviceSegmentedReduce` e `thrust::reduce_by_key`, apresentados na Seção 3.

Inicialmente, será discutido que parâmetro é adequado para determinar o nível de granularidade mais eficiente para cada tamanho de segmento. Para essa análise, serão gerados conjuntos de dados com 16 milhões de elementos, com a quantidade de segmentos variando de 2 a 8 milhões. Em seguida, os algoritmos de redução segmentada serão analisados em dois cenários: O primeiro cenário analisa o desempenho dos algoritmos quando os segmentos são regulares, ou seja, possuem tamanhos fixos, como ocorre quando a redução é aplicada em linhas de uma matriz. O segundo cenário investiga o desempenho dos algoritmos quando os segmentos possuem tamanhos variados.

Para gerar conjuntos de dados com segmentos irregulares, foram criados vetores de tamanhos aleatórios utilizando duas distribuições: distribuição exponencial com valor de $\lambda = 4$ e distribuição normal com tamanho de segmentos médio t, conforme a Equação 1, e desvio padrão $t/2$ no tamanho dos segmentos.

$$t = \frac{\text{quantidade de elementos}}{\text{quantidade de segmentos}} \tag{1}$$

Nessas análises, são utilizados conjuntos de dados com 32 milhões de elementos, com a quantidade de segmentos variando de 2 a 16 milhões. Todos os experimentos foram repetidos 30 vezes, e a vazão média foi reportada. Intervalos de confiança de 95% também foram calculados, mas, como nenhum resultado apresentou variação significativa em relação à média, esses intervalos não foram reportados. As reduções foram realizadas utilizando a operação de máximo, com dados do tipo inteiro sem sinal (*unsigned integer*).

A máquina utilizada nos experimentos possui processador Intel Xeon Silver 4314 @ 2.40GHz com 16 núcleos (32 *hyperthreads*), 32GB de RAM e GPU NVIDIA A4500 que possui 56 multiprocessadores e 7168 núcleos CUDA. O sistema operacional utilizado foi o Linux Ubuntu 20.04.6 LTS e as implementações foram compiladas com a versão 12.4 da biblioteca CUDA. Foi utilizada a versão 2.3 tanto do CUB quanto do Thrust.

6. Resultados e Discussões

Os resultados da comparação entre as implementações de redução não segmentada estão apresentados na Tabela 1. A análise da tabela revela que a implementação `ui32x4` obteve o melhor desempenho entre todas as implementações avaliadas. Além disso, a

implementação ui32_atomic_op apresentou uma vazão superior às implementações ui32_sh_mem e ui32_shfl em todos os testes realizados. Isso indica que, para a GPU utilizada nos experimentos, as operações atômicas se mostraram a forma mais eficiente para agregar os resultados das *threads* de um bloco. Portanto, o desempenho superior da ui32x4 pode ser atribuído ao uso de operações atômicas para a agregação de dados, além da leitura vetorizada para acessar os dados do conjunto de entrada.

	Quantidade de elementos									
	128K	256K	512K	1M	2M	4M	8M	16M	32M	64M
Thrust	6.34	11.90	22.70	38.77	59.01	83.93	106.30	122.97	134.17	140.46
CUB	20.93	34.73	61.88	86.80	104.11	122.60	133.39	140.68	144.55	146.68
ui32_sh_mem	30.93	49.94	73.13	93.13	109.79	123.72	131.99	136.92	139.89	142.08
ui32_atomic_op	36.52	58.97	81.57	98.86	113.72	126.22	133.54	137.74	140.36	142.36
ui32_shfl	36.04	57.54	80.09	98.30	112.99	125.79	133.34	137.65	140.28	142.40
ui32x4	**36.97**	**59.53**	**84.37**	**105.83**	**122.00**	**133.41**	**140.05**	**144.09**	**146.18**	**147.45**
speedup (CUB/ui32x4)	**1.77**	**1.71**	**1.36**	**1.22**	**1.17**	**1.09**	**1.05**	**1.02**	**1.01**	**1.01**

Tabela 1. Vazão (Giga elementos/s) das implementações de redução não segmentada para conjuntos de dados com diferentes tamanhos.

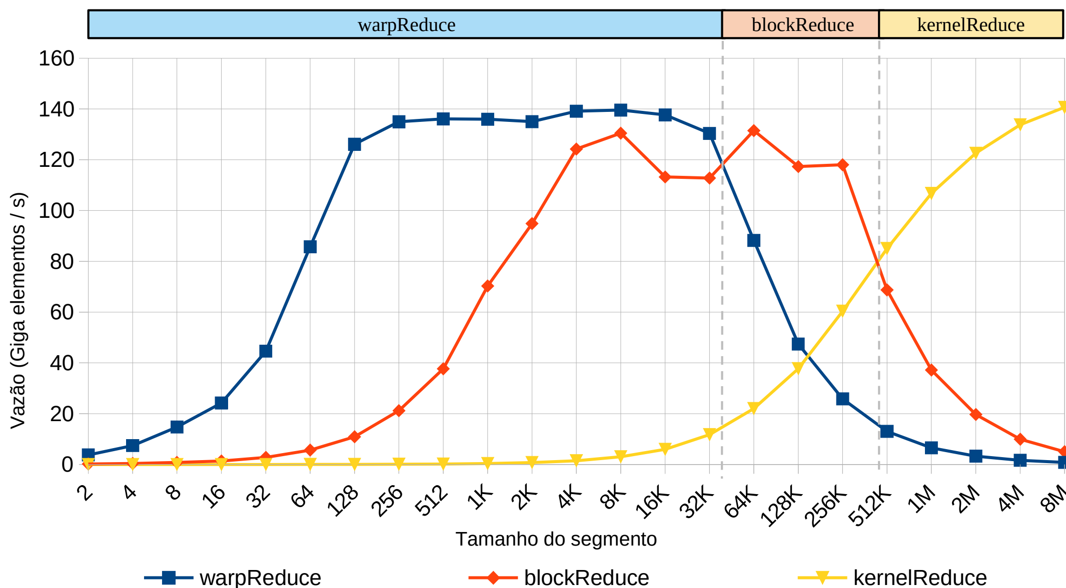

Figura 1. Desempenho dos diferentes níveis de granularidade para conjuntos de dados com variados tamanhos de segmentos.

Os resultados dos testes avaliando os níveis de granularidade estão apresentados na Figura 1. A partir da análise do gráfico, pode-se concluir que o warpReduce apresenta desempenho superior para segmentos de até 32 mil elementos. Para tamanhos maiores, a granularidade por bloco torna-se mais eficiente, até atingir aproximadamente 512 mil elementos. A partir desse ponto, a redução por *kernel* se revela a mais eficaz. Com base nesses resultados, os parâmetros BEST_WARP_REDUCE e BEST_BLOCK_REDUCE foram definidos como 33 mil e 500 mil, respectivamente. O bestReduce foi compilado com esses parâmetros para os testes subsequentes.

A Figura 2 apresenta os resultados dos testes comparando os diferentes algoritmos

de redução segmentada. Inicialmente, observa-se que o `bestReduce` apresenta desempenho significativamente superior aos demais algoritmos quando o tamanho dos segmentos é elevado. Isso se deve à eficiência do algoritmo `ui32x4`, utilizado para redução em segmentos maiores que 500 mil elementos. O `bestReduce` também supera o algoritmo da biblioteca CUB quando o tamanho dos segmentos é reduzido. Isso ocorre porque o CUB utiliza um bloco por segmento, o que pode levar a *threads* ociosas quando os segmentos são pequenos, enquanto o `bestReduce` emprega *warps* para realizar a redução, reduzindo consideravelmente a quantidade de *threads* ociosas.

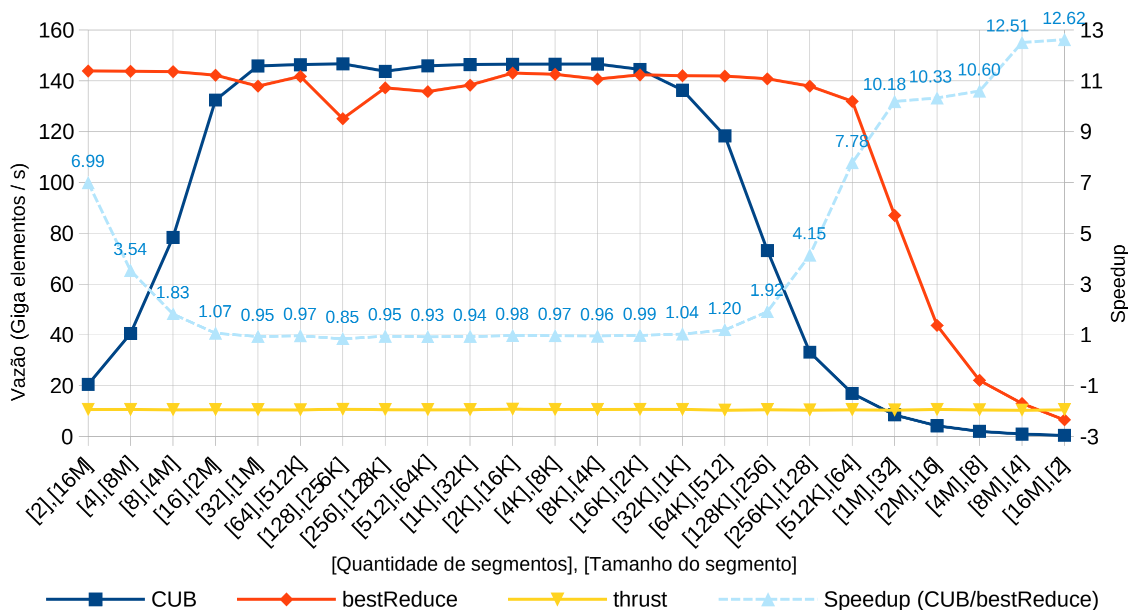

Figura 2. Desempenho dos algoritmos de redução segmentada para conjuntos de dados com 32 milhões de elementos e segmentos de tamanhos fixos. A linha azul tracejada indica o *speedup* do `bestReduce` em relação ao algoritmo da biblioteca CUB.

No entanto, o algoritmo da biblioteca CUB apresenta melhor desempenho em segmentos de tamanho médio. O resultado inferior em relação ao CUB pode ser explicado pelo *overhead* de ter que escolher a melhor estratégia em tempo de execução. Mesmo assim, o desempenho do `bestReduce` se mantém próximo a 95% do desempenho do CUB, com uma aceleração de até 12.62 vezes no melhor cenário. Quanto ao algoritmo Thrust, a vazão permanece praticamente constante, uma vez que a redução por chave utilizada pelo Thrust não é afetada pela quantidade ou pelo tamanho dos segmentos. Uma vantagem dessa abordagem é que o Thrust consegue um bom desempenho em comparação aos algoritmos analisados quando a quantidade de segmentos é muito grande.

Um aspecto a ser notado é a pequena queda no desempenho quando os segmentos têm tamanho de 256 mil elementos. Isso ocorre porque, para esse tamanho, a abordagem blockReduce é selecionada. Nesse algoritmo, são lançados 2 blocos de threads por multiprocessador, sendo que a GPU utilizada possui 56 multiprocessadores. Assim, podem existir no máximo 112 blocos executando simultaneamente na GPU. No entanto, como é necessário processar 128 segmentos, 112 segmentos são reduzidos em paralelo e, em seguida, os 16 restantes. Como o número de segmentos não é um múltiplo da quantidade de blocos de threads, apenas 16 blocos estarão em execução ao mesmo tempo, resultando

em uma subutilização dos recursos da GPU e, consequentemente, em uma queda na vazão do algoritmo para essa quantidade de segmentos. Essa queda de desempenho não ocorre para segmentos de tamanho 512 mil, pois, nesse caso, é utilizado o `kernelReduce`. Isso sugere que pode haver um algoritmo mais eficiente que combine as abordagens do `blockReduce` e do `kernelReduce` para otimizar o desempenho nesse cenário.

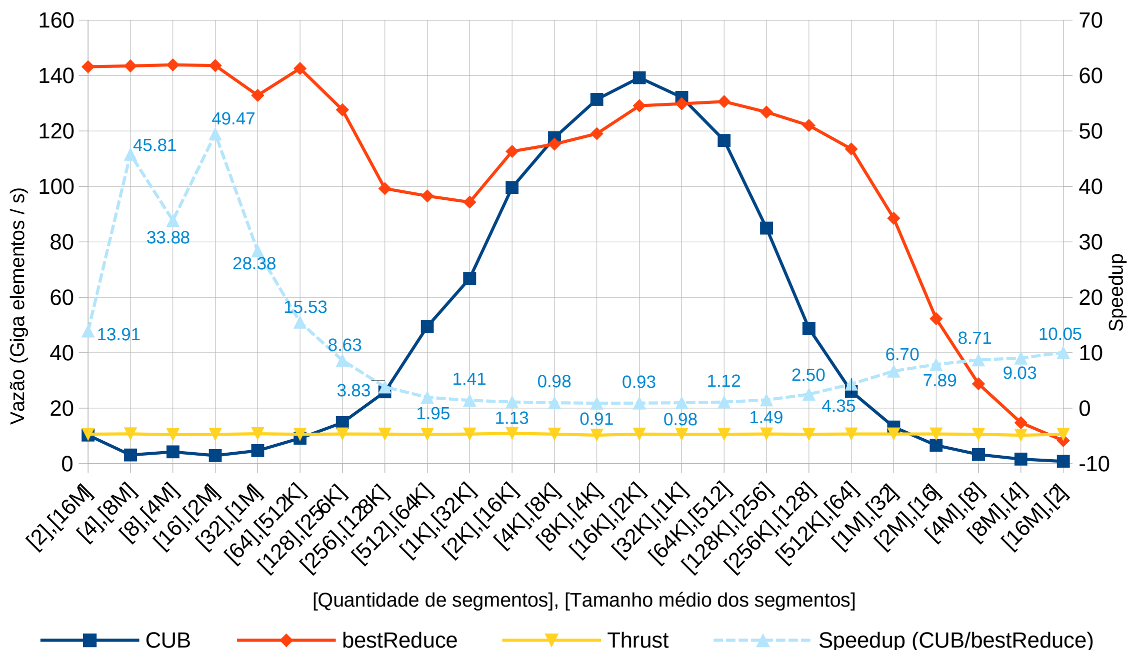

Figura 3. Desempenho dos algoritmos de redução segmentada para conjuntos de dados com 32 milhões de elementos com segmentos cujos tamanhos foram gerados a partir da distribuição normal. A linha azul tracejada indica o *speedup* do `bestReduce` em relação ao algoritmo da biblioteca CUB.

Os resultados dos experimentos comparando os algoritmos de redução segmentada com segmentos de tamanhos irregulares estão apresentados nas Figuras 3 e 4. Em comparação com o CUB, o `bestReduce` demonstrou desempenho superior na maioria dos testes, alcançando aceleração de até 49.47 vezes quando o conjunto de dados possui 16 segmentos, como pode ser visto na Figura 3. Isso destaca a eficácia da estratégia do `bestReduce` de selecionar dinamicamente a melhor forma de dividir o trabalho entre as *threads* em tempo de execução. Enquanto o CUB utiliza um bloco de *threads* por segmento [Larsen and Henriksen 2017], o que pode resultar em uma carga de trabalho desbalanceada, o `bestReduce` consegue manter a carga de trabalho mais equilibrada entre as *threads*. No entanto, o CUB ainda possui melhor desempenho quando o tamanho médio dos segmentos está em um determinado intervalo, como também foi observado na Figura 2. Uma forma de aprimorar este trabalho seria instanciar o CUB para executar segmentos no intervalo de tamanhos em que ele tem melhor performance. Dessa maneira, a biblioteca do bestReduce obteria o melhor desempenho em todas as faixas de tamanhos.

Por fim, o Thrust apresentou uma vazão constante independentemente da quantidade de segmentos, como esperado, uma vez que sua abordagem de redução por chave não é influenciada pelo tamanho ou pela quantidade dos segmentos.

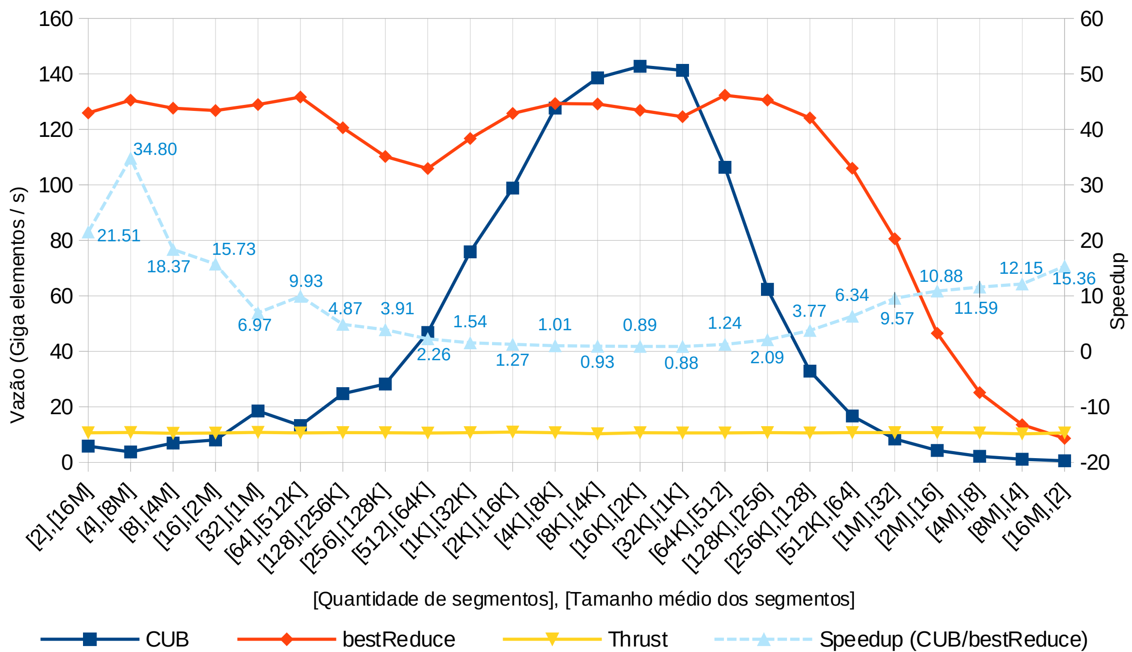

Figura 4. Desempenho dos algoritmos de redução para conjuntos de dados com 32 milhões de elementos com segmentos cujos tamanhos foram gerados a partir da distribuição exponencial. A linha azul tracejada indica o *speedup* do `bestReduce` em relação ao algoritmo da biblioteca CUB.

7. Conclusões

A operação de redução combina todos os elementos de uma coleção aplicando uma operação binária, como soma, máximo ou mínimo, para obter um único valor resultante. Uma variação dessa operação é a redução segmentada, cujo objetivo é aplicar a redução a cada segmento de um vetor segmentado. Este trabalho investigou estratégias de otimização para redução segmentada e não segmentada em GPUs. Embora as técnicas existentes de redução segmentada sejam consistentes, elas frequentemente apresentam desempenho subótimo em termos absolutos. Essas técnicas são tipicamente otimizadas para cargas de trabalho específicas, o que pode resultar em degradação de desempenho para certos conjuntos de dados, especialmente quando os tamanhos dos segmentos variam.

As estratégias de otimização para redução não segmentada apresentaram melhorias em relação aos algoritmos disponíveis, superando o algoritmo de redução da biblioteca CUB e Thrust e outras bibliotecas em diversos cenários. Este artigo também propôs um algoritmo para redução segmentada que utiliza três estratégias distintas para lidar com tamanhos variados de segmentos e é capaz de selecionar a melhor estratégia em tempo de execução, otimizando o processamento de cada segmento conforme seu tamanho. O algoritmo demonstrou um desempenho consistente, alcançando uma aceleração de até 49.47 vezes para redução segmentada em segmentos de tamanhos irregulares, até 12.62 vezes para segmentos de tamanhos fixos, e até 1.77 vezes para redução não segmentada, em comparação aos algoritmos de redução da biblioteca CUB.

O algoritmo desenvolvido neste artigo estará disponível em: https://github.com/MichelBC/bestReduce.git

Agradecimentos

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

Referências

Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J. W., Lee, S.-H., and Skadron, K. (2009). Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE international symposium on workload characterization (IISWC)*, pages 44–54. IEEE.

Cordeiro, M. and Zola, W. (2023). KNN paralelo em GPU para grandes volumes de dados com agregação de consultas. In *Anais do XXIV Simpósio em Sistemas Computacionais de Alto Desempenho*, pages 253–264, Porto Alegre, RS, 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 *2012 Innovative Parallel Computing (InPar)*.

Harris, M. (2007). Optimizing parallel reduction in CUDA. *Nvidia developer technology*.

Johnson, J., Douze, M., and Jégou, H. (2019). Billion-scale similarity search with gpus. *IEEE Transactions on Big Data*, 7(3):535–547.

Jradi, W. A. R., do Nascimento, H. A. D., and Martins, W. S. (2018). A fast and generic gpu-based parallel reduction implementation. In *2018 Symposium on High Performance Computing Systems (WSCAD)*, pages 16–22. IEEE.

Larsen, R. W. and Henriksen, T. (2017). Strategies for regular segmented reductions on gpu. In *Proceedings of the 6th ACM SIGPLAN International Workshop on Functional High-Performance Computing*, pages 42–52.

Luitjens, J. (2014). Faster parallel reductions on kepler. `http://devblogs.nvidia.com/parallelforall/faster-parallelreductions-kepler`.

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. Association for Computing Machinery.

NVIDIA (2024). CUB. `https://nvidia.github.io/cccl/cub/`.

Thrust (2024). Thrust: The C++ Parallel Algorithms Library. `https://docs.nvidia.com/cuda/thrust/index.html`.

Warashina, T., Aoyama, K., Sawada, H., and Hattori, T. (2014). Efficient k-nearest neighbor graph construction using mapreduce for large-scale data sets. *IEICE TRANSACTIONS on Information and Systems*, 97(12):3142–3154.

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