Redução de Complexidade de Tempo em GPUs

Ricardo Nobre* Tiago Carneiro* Marcos Negreiros* Felipe Martins Muller#

*Universidade Estadual do Ceará. #Universidade Federal de Santa Maria

Resumo

Este artigo aborda a questão da construção de algoritmos paralelos e avaliação dos resultados a partir da redução de complexidade obtida pelo emprego massivo do paralelismo, em contraponto a obtenção de *speedups* como delineadores da construção de algoritmos paralelos. Mostra-se que, em um problema simples de pesquisa em um vetor, é mais proveitosa a abordagem da redução de complexidade com GPUs.

Palavras-chaves: Redução de complexidade, paralelismo, *speedup,* problema da pesquisa, GPU.

Contato dos Autores:
ricardo.nobre@serpro.gov.br;carneiro@larces.uece.br;negreiro@graphvs.com.br;*felipe*@inf.*ufsm*.br

1. Introdução

A cada dia um novo artigo é publicado sobre a aplicação massiva de paralelismo utilizando GPUs para resolver os mais diversos tipos de problemas, nas mais diversas áreas. Uma das principais razões para a adoção desta tecnologia é o seu baixo custo [Vineet *et al*. 2010] e o potencial de substituir um grande número de computadores superescalares em determinados tipos de aplicação [Schaa e Kaeli, 2009].

Há uma onda vibrante e frenética de pesquisas utilizando *GPU computing* na busca por computações com resultados cada vez melhores. Contudo como devemos medir estes resultados? Quão eficiente é o algoritmo paralelo construído? Segundo Grama *et al*. [2003] várias são as métricas utilizadas para analisar um algoritmo paralelo, destacando-se: tempo de execução; *overhead*; *speedup*; eficiência e custo.

Nas pesquisas atuais ligadas a *GPU Computing* há um enfoque na exposição do *speedup* como métrica para avaliação de eficiência do algoritmo [Boyer *et al*. 2010; Harris *et al*. 2009; Vineet *et al*. 2010], pouco referenciado-se a redução de complexidade de tempo como fator essencial para obtenção de excelentes resultados.

Para Akl [1989], o tempo de execução é a medida mais importante na análise de algoritmos paralelos, devendo-se, ao se avaliar um algoritmo, levar em consideração duas importantes questões: (i) – Ele é o algoritmo mais rápido possível para o problema? (ii) – Se não, como ele se compara com outros algoritmos existentes para o mesmo problema?

A primeira questão é respondida comparando-se a complexidade de tempo do algoritmo com o limite inferior do melhor algoritmo conhecido para o mesmo problema, no pior caso. A segunda é respondida através da comparação do tempo de execução do novo algoritmo, com o limite superior para o problema.

Neste contexto, os projetistas deveriam estar também focados em reduzir a complexidade dos algoritmos paralelos construídos, uma vez que o tipo de abordagem levará inexoravelmente a obtenção de ótimos *speedups*, bem como em apresentar os resultados de suas pesquisas acrescentando esta importante métrica, como fizeram Harish e Narayanan [2007] e Buluc *et al*. [2009].

Para demonstrar que esse pensamento é plausível abordamos a seguir um problema simples (pesquisa em um vetor), para se ter um melhor entendimento da construção de métodos, onde o foco principal no projeto do algoritmo será a redução de complexidade de tempo do mesmo, independente das questões arquiteturais conhecidas das GPUs e que causam impactos negativos na computação do tempo, como é o caso do uso de memória global, e outras.

Este artigo está organizado como segue: na seção 2 discutimos o problema computacional clássico da Pesquisa em um vetor, na seção 3 discutimos a redução da complexidade de tempo deste problema, na seção 4 discutimos os resultados computacionais usando *GPU Computing*.

2. O problema da Pesquisa

O problema consiste em achar um determinado elemento $\boldsymbol{x}$ em uma lista $\boldsymbol{L}$ de n elementos ordenados ou não ordenados, onde $\boldsymbol{L} = (a_1, a_2, ..., a_n)$, ou mais precisamente, dado um número $\boldsymbol{x}$ em uma lista $\boldsymbol{L}$ qualquer, encontrar um índice $\boldsymbol{j}$ tal que $\boldsymbol{L}[\boldsymbol{j}] = \boldsymbol{x}$.

```
procedure Pesquisa( L[], x, n )
1. begin
2.     j := 0; k := -1;
3.     while (j < n) and (k = -1)
4.         if L[j] = x then
5.             k := j ;
6.         else j := j + 1;
7.         endif
8.     endwhile
9.     return k
10. end
```

Algoritmo 1 – Algoritmo de Pesquisa Sequencial

Para efeito de análise será abordado apenas o caso em que os elementos da lista ***L*** estão desordenados, ***L*** é composta apenas de inteiros não negativos e que não será empregado nenhum algoritmo para ordenar ***L***. Neste cenário, um algoritmo simples para encontrar um elemento qualquer em uma lista ***L*** desordenada percorre a lista do primeiro ao último. O Algoritmo 1 realiza este processo de forma sequencial, possuindo complexidade de tempo igual a *O* (*n*), para o pior caso.

3. Reduzindo Complexidade de Tempo

Para construção do algoritmo paralelo para o problema da pesquisa utilizamos uma técnica de busca em blocos de elementos, no qual um conjunto de ***p*** processadores, fará uma pesquisa em um bloco de tamanho ***b***, onde ***b*** = ***p***, em ***d*** iterações, sendo $d = \lceil \log n \rceil$, onde:

$$p = \begin{cases} \left\lceil \frac{n}{\log n} \right\rceil, \ para\ n > 1 \\ 1, caso\ contrário \end{cases} \qquad (1)$$

sendo ***p*** uma função de qualidade $f_p(n)$ a qual determina o número de unidades de processamento (*threads*) utilizadas e que garantirão a redução de complexidade pretendida, em uma lista de ***n*** elementos.

A Figura 1 mostra a lista ***L*** = {x_0,...,x_8}, com ***n*** = 8, buscando-se o valor localizado em x_6. Como a lista tem 8 elementos serão necessárias 3 *threads* para varrer a lista em *log 8* passos.

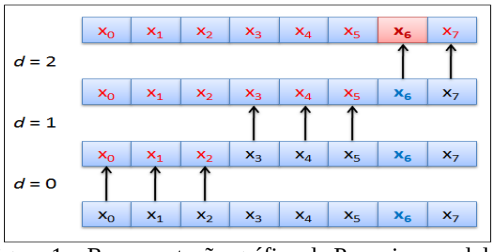

Figura 1 – Representação gráfica da Pesquisa paralela.

O Algoritmo 2 realiza o processo de pesquisa, particionando a lista ***L*** em blocos de elementos de tamanho ***b***, onde cada *thread* irá verificar um elemento do bloco. A variável ***k*** é compartilhada por todas as *threads*, tendo sido atribuído o valor (-1) a ela, indicando que o elemento não foi encontrado. Assim, quando alguma *thread* encontrar o valor que está sendo procurado, esta variável é modificada, indicando em que posição da lista ***L*** ele se encontra, forçando todas as demais *threads* a cessarem a busca, conforme linhas 5 e 8. O processo irá ser realizado, no pior caso, até que toda a lista seja verificada ou até o elemento ser encontrado, ou seja, em ***log n*** passos.

Em CUDA executamos o Algoritmo 2 invocando *pesquisa_paralelo* <<<*bthreads, maxthreads*>>>, onde a quantidade de blocos de *threads* é obtida por:

$$bthreads = (p + maxthreads + 1)/maxthreads \qquad (2)$$

sendo *maxthreads* a quantidade máxima de *threads* existente em um bloco de *threads*.

```
procedure  Pesquisa_Paralelo ( L[], x, n, b, k )
1.  begin
2.    for d := 0 to ⌈log n⌉ do
3.      for all j := (b * d) to (b * (d + 1) - 1) in parallel do
4.        if k ≠ -1 then  // elemento 'x' encontrado
5.          break;  // sair do laço
6.        endif
7.        if L[j] = x then  // elemento 'x' encontrado
8.          k := j ; d := n; break;
9.        endif
10.     endfor
11.   endfor
12.   return k
13. end
```
Algoritmo 2 – Algoritmo de Pesquisa Paralelo

Para efeitos de avaliação de tempo, não será empregada nenhuma técnica de otimização de memória, gerenciamento de conflitos e outros, valendo-se apenas do emprego massivo do paralelismo para redução da complexidade de tempo. No entanto, como não é comum este tipo de abordagem em artigos relacionados à *GPU Computing*, serão apresentados vários aspectos ligados ao tempo de execução, dentre eles o tempo de execução no *Kernel*, o tempo em multi GPUs, e o tempo computacional total, incluindo-se neste último, além do tempo consumido no *kernel*, o *overhead* relacionado à leitura dos dados, envio dos dados entre CPU para GPU e vice-versa, tempo de sincronismo, tempo de comunicação, alocação e desalocação de memória na CPU e GPU, entre outros.

3.1. Variação Multi-GPU

Como o objetivo do presente artigo é mostrar que a redução da complexidade de tempo, através do uso massivo do paralelismo conduzirá obtenção de excelentes resultados, faz-se necessário garantir que haverá uma quantidade de unidades de processamento ***p*** suficientemente grande em relação ao tamanho da entrada ***n***, tal que ***p*** << ***n***. Se ***p*** não for suficientemente grande, deve-se adicionar outras GPUs de forma a garantir aquela condição.

Uma pequena, mas substancial alteração deverá ser realizada no processo de carga dos dados da CPU para as GPUs e destas, quando do final do processamento, para a CPU. A Figura 2 mostra o processo em duas fases. Na primeira fase a CPU irá dividir os dados de maneira balanceada entre as duas GPUs, conforme item (a). Uma vez feito o balanceamento de carga os dados serão enviados para as GPUs envolvidas de maneira assíncrona. Desta forma não é necessário que a cópia dos dados para a GPU 1 tenha terminado para ser iniciada a cópia dos dados para a GPU 2, conforme item (b). A partir de então o processamento será iniciado, também de maneira assíncrona, nas GPUs, ou

seja, uma vez que os dados da GPU 1 tenham sido copiados ela iniciará sua execução, não sendo necessário que ela aguarde que os dados da GPU 2 tenha sido completamente copiados, e vice-versa. Durante a execução, cada GPU irá armazenar o resultado da computação em uma variável única, não compartilhada, no caso, R_1 para a GPU 1 e R_2 para a GPU 2, conforme item (c). Uma vez tendo sido computadas as tarefas na GPU 1 e 2 os dados serão copiados para a CPU de maneira assíncrona, semelhante ao processo descrito no item (b). Este processo está descrito em (d), sendo que a cópia dos dados da GPU 1 para a CPU não precisa esperar pela conclusão das tarefas que estão sendo executadas pela GPU 2 e vice-versa. As equações descritas em (1) e (2) permanecem inalteradas, porém com n=n/2 para cada GPU, em razão da divisão dos dados.

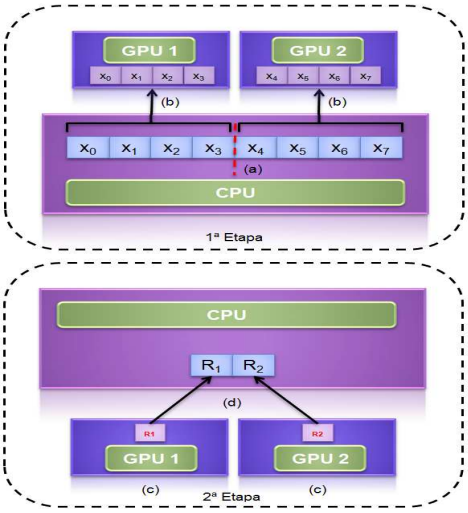

Figura 2 – Divisão dos dados e retorno dos resultados.

4. Avaliando os resultados

Foram realizados testes utilizando uma GeForce GT 240, com 96 processadores, 1371 MHz de *clock* de processador, 1700 MHz de *clock* de memória, com 1 GB de memória, duas GeForce GTX 580, com 512 processadores, 1544 MHz de *clock* de processador, 2004 MHz de *clock* de memória, e 3GB de memória e um Intel Core i7 2600 (3.4 GHz), com 8 núcleos e Ubutun GNU Linux 11.10 OS (*kernel* 3.0.1). A arquitetura *GPU Computing* utilizada foi o CUDA 4.2, sendo as implementações construidas em C, incluindo o gerador das instâncias aplicadas nos testes. Os algoritmos foram testados considerando o pior caso, ou seja, o elemento procurado não existia no vetor.

O primeiro teste a ser realizado foi referente ao comportamento do tempo dentro do *kernel* da GPU, ou seja, o tempo de computação propriamente dito, não sendo levados em consideração: a cópia dos dados da CPU para GPU e vice-versa, alocação e desalocação de memória na GPU e outros.

Conforme NICKOLLS e DALLY [2010] as GPUs são dotadas de unidades de processamento simultâneos (UPS) em cada *streaming multiprocessors* (SM), cuja função é minimizar o tempo de latência da memória. Assim, uma vez atingida esta capacidade, as *threads* passarão a se alternar na execução das tarefas, o que levará ao aumento do tempo computacional no *kernel*.

Os testes apresentados na Figura 3 apresentam uma inflexão na função que descreve o tempo real justamente no ponto onde a quantidade UPS é atingida]65.536, 131.072[.

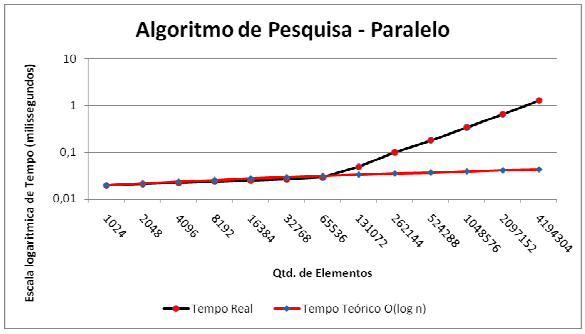

Figura 3 – Tempo no *Kernel* com uma GPU - GT 240, com 6.411 - UPS

Assim, em uma GPU com uma maior quantidade de UPS este ponto será deslocado para a direita do gráfico, conforme Figura 4. Além do deslocamento do ponto de saturação para o intervalo]262.144, 524.288[com uma GPU e]524.288, 1.048.576[com duas GPUs, verifica-se a redução do tempo de execução em 50%.

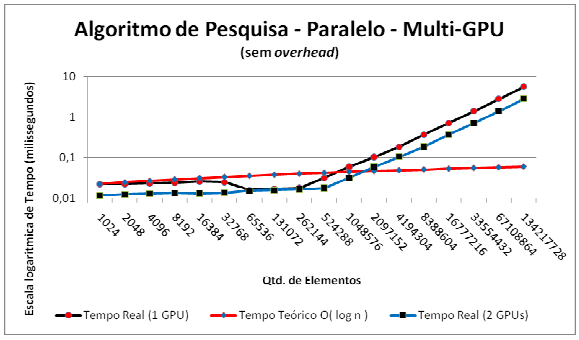

Figura 4 – Tempo no *Kernel* com duas GPUs (GTX 580), cada uma com 24.576 UPS.

Uma vez verificado o tempo de execução consumido pelas computações dentro do *kernel* passaremos a análise do impacto relacionado ao tempo computacional total, incluindo-se o *overhead*. A idéia é verificar o comportamento do tempo com a utilização de mais de uma GPU, bem como comparar estes tempos com o tempo do melhor algoritmo seqüencial que resolve o mesmo problema.

A Figura 5 mostra uma pequena perda de desempenho (4,7% em média) relacionada ao processo de divisão dos dados entre GPUs, contradizendo o ganho obtido no tempo computacional no *kernel*, visto na Figura 4. Esta discrepância é resultado da dominância do tempo de *overhead* sobre o tempo de computação propriamente dito (tempo no *kernel*).

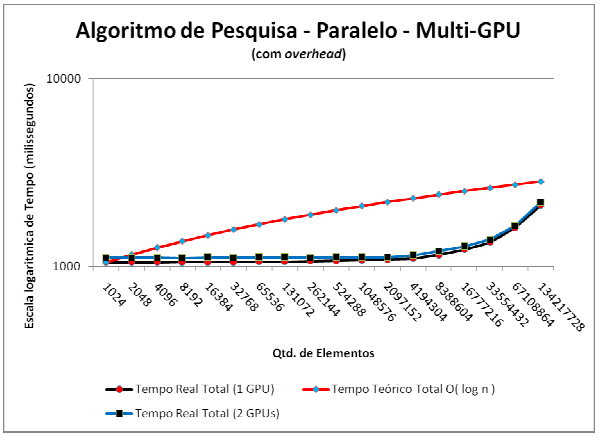

Figura 5 –Desempenho do Algoritmo de Pesquisa Paralelo com uma e duas GPUs (GTX 580) – com *overhead*

Como o intuito é verificar se a estratégia da adoção da redução de complexidade de tempo irá conduzir a bons resultados, incluindo nestes o *speedup* foi realizado um comparativo entre o tempo do algoritmo paralelo (T_p), e o tempo do melhor algoritmo seqüencial (T_s), como mostrado na Figura 6.

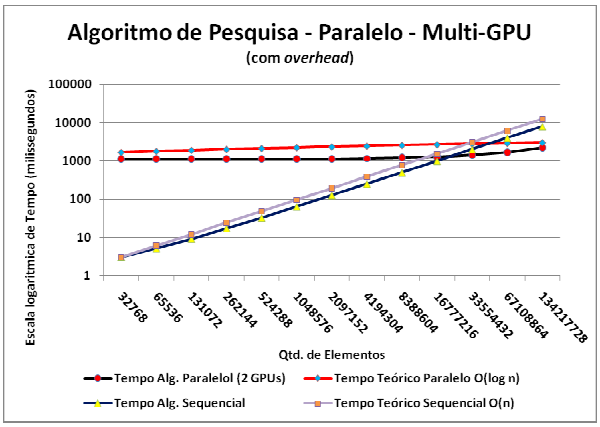

Figura 6 – GTX 580 versus Intel Core i7

Pode-se verificar, de forma inequívoca, que o crescimento do algoritmo paralelo é $O(log\ n)$, enquanto que o do seqüencial é $O(n)$, e que para uma entrada assintótica, independente de otimizações que podem ser realizadas nos códigos para resolver problemas relacionados a arquitetura GPU, a abordagem paralela terá os melhores resultados de tempo a partir de um certo ***n***, sendo o crescimento do *speedup* ($fs = T_s/T_p$) garantido pela redução de complexidade de tempo obtida na construção do algoritmo paralelo.

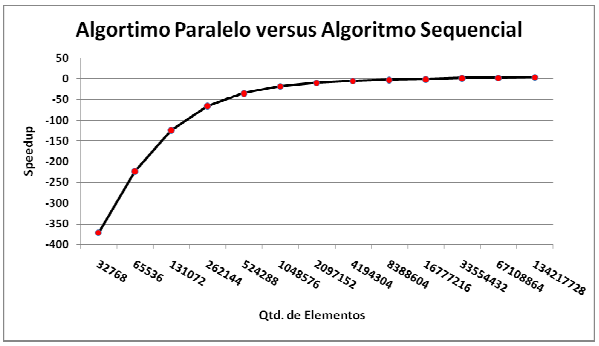

Figura 7 – *Speedup*: Algoritmo Paralelo e Seqüencial

A Figura 7 mostra a função de *speedup*, denotando que a partir do intervalo]16777216, 33554432[o algoritmo paralelo apresentou melhores resultados que o sequencial, sendo, no último valor computado (134217728), 3,7x mais rápido. A recuperação do *speedup*, que pode ser melhor visualizada no intervalo]32768, 33554432[é oriunda da melhor complexidade de tempo do algoritmo paralelo.

5. Conclusão

A estratégia de construção de algoritmos paralelos baseados na redução de complexidade de tempo como fator motivador para obtenção de excelentes resultados, utilizando uma arquitetura massivamente paralela, apresentou resultados satisfatórios para uma aplicação simples, mas cujo tempo dominante está relacionado ao *overhead*. Neste aspecto o algoritmo paralelo manteve seu crescimento delineado pela função teórica da complexidade de tempo, o que o fez superar o tempo do melhor algoritmo seqüencial a partir de um determinado ***n***, isto porque a função teórica de complexidade do tempo do algoritmo serial é maior que aquela.

Reduzimos também a complexidade do problema do Maior e Menor elemento em uma lista e no momento estamos trabalhando com o problema da mochila (0-1).

Referências

AKL, SELIM G., 1989. THE DESIGN AND ANALYSIS OF PARALLEL ALGORITHMS. NJ: PRENTICE-HALL, INC., UPPER SADDLE RIVER.

BOYER, V.; BAZ, D.E.; ELKIHEL, M., 2010. DENSE DYNAMIC PROGRAMMING ON MULTI GPU. TECHNICAL REPORT LASS-CNRS NO. 10509.

BULUC, A.; GILBERT, J. R.; BUDAK, C., 2009. SOLVING PATH PROBLEMS ON THE GPU, PARALLEL COMPUTING, NO. 7000012980.

HARISH, P.; NARAYANAN, P. J., 2007. ACCELERATING LARGE GRAPH ALGORITHMS ON THE GPU USING CUDA. IN HIPC.

HARRIS, MARK; GARLAND, MICHAEL; SATISH, NADATHUR., 2009. DESIGNING EFFICIENT SORTING ALGORITHMS FOR MANYCORE GPUS. PROCEEDINGS OF THE 2009 IEEE INTERNATIONAL SYMPOSIUM ON PARALLEL & DISTRIBUTED PROCESSING, 1-10.

NICKOLLS, JOHN; DALLY, WILLIAN J., 2010. THE GPU COMPUTING ERA. IEEE MICRO, 30(2), 56–69.

SCHAA, D., AND KAELI, D., 2009. EXPLORING THE MULTIPLEGPU DESIGN SPACE. IN PROC. OF THE IEEE (IPDPS), 1-12.

VINEET, VIBHAV; HARISH, PAWAN; PATIDAR, S.; NARAYANAN, P. J., 2009. FAST MINIMUM SPANNING TREE FOR LARGE GRAPHS ON THE GPU. ACM SIGGRAPH/EUROGRAPHICS SYMPOSIUM ON HIGH PERFORMANCE GRAPHICS.