Implementação de um algoritmo paralelo na GPU para o Problema da Máxima Subsequência Crescente Permitindo Inversões e Rotações

Matheus Nogueira, David Pena, Anolan Milanés
Departamento de Engenharia da Computação – CEFET-MG
Av. Amazonas, 7675 – Nova Gameleira – Belo Horizonte - MG - Brasil
Email: {matheus, david, anolan}@decom.cefetmg.br

Sebastián Urrutia
Departamento de Ciência da Computação – Universidade Federal de Minas Gerais
Av. Antônio Carlos, 6627 – Pampulha – Belo Horizonte – MG CEP 31270-901
surrutia@dcc.ufmg.br

RESUMO

Este trabalho aborda o problema de encontrar o valor máximo de uma função sobre o conjunto de permutações dos n primeiros números naturais. A função computa o valor mínimo do tamanho da maior subsequência crescente dentre todas as rotações e inversões de uma permutação π. Esta função surge como subproblema em alguns problemas de otimização combinatória. A dificuldade deste problema é que o aumento do valor de n provoca um aumento exponencial do número de permutações. Esse trabalho propõe uma abordagem de processamento exaustivo implícito dessas permutações. Apesar disso, o problema continua crescendo de forma exponencial e se torna computacionalmente intratável na prática para valores pequenos de n. Isto motiva a utilização de técnicas de paralelismo na implementação do algoritmo. Este artigo apresenta e compara duas abordagens para a implementação paralela do algoritmo utilizando Unidades de Processamento Gráfico(GPU).

PALAVRAS CHAVE. Otimização combinatória, permutações, paralelismo, GPU, CUDA.

Área Principal: Otimização combinatória, paralelismo

ABSTRACT

This paper addresses the problem of finding the maximum value of a function on the set of permutations of the first n natural numbers. The function computes the minimum value of the length of the longest subsequence, among all rotations and inversions of a permutation π. The computation of this function appears as sub-problem in some combinatorial optimization problems. The difficulty of the problem relies on the fact that increasing the value of n causes an exponential increase in the number of permutations. This paper proposes an implicit comprehensive approach for the processing of these permutations. Nevertheless, the problem still grows exponentially and becomes computationally intractable in practice for small values of n. This motivates the use of parallelism techniques in the implementation of the algorithm. This article presents and compares two parallel approaches of the algorithm using graphics processing units (GPU).

KEYWORDS. Combinatorial Optimization, Permutations, Paralelism, GPU, CUDA.

Main Area: Combinatorial Optimization, paralelism

1. Introdução

Em vários problemas de otimização combinatória, as soluções podem ser representadas por permutações dos n primeiros números naturais. Em problemas de sequenciamento, deve-se encontrar uma ordem de execução das tarefas de forma a otimizar certa função objetivo. Um exemplo típico deste tipo de problema é o Caixeiro Viajante (CV).

Algumas variações do CV consideram que dois grupos disjuntos de tarefas relacionadas devem ser feitas em ordens compatíveis. No caso do Caixeiro Viajante com Múltiplas Pilhas (Petersen e Madsen 2009), a compatibilidade de duas permutações depende do número cromático do grafo de permutação associado a estas (Casazza, Ceselli e Nunkesser 2012). Por sua vez, ao considerar uma das permutações fixa e igual à sequência dos primeiros n números naturais, esse número equivale ao tamanho da maior subsequência crescente, não necessariamente contínua, existente na outra permutação (Golumbic 2004).

O problema da maior subsequência crescente (LIS, do inglês *Longest Increasing Sequence*) é bastante estudado na literatura e se conhece um algoritmo $O(n * \lg n)$ para seu processamento (Romik 2015). Um problema equivalente a este corresponde ao cômputo da maior subsequência decrescente (LDS, do inglês *Longest Decreasing Sequence*).

Para limitar superiormente o possível grau de incompatibilidade de duas permutações, deseja-se saber qual é o máximo valor possível da maior subsequência crescente de uma permutação de n elementos quando certas modificações são permitidas na permutação para tentar reduzir esse valor.

Claramente, se nenhuma operação é permitida na permutação, o valor procurado é igual a n, dado que a própria sequência dos primeiros números naturais é crescente. Quando a inversão da permutação é permitida, existirão permutações nas quais, tanto na própria permutação quanto na sua inversão, haverá subsequências crescentes de tamanho não menor que $\lceil n/2 \rceil$, e este é o máximo valor possível. Provou-se também que quando rotações da permutação são permitidas, as permutações com maior tamanho de subsequências crescentes em todas suas rotações tem valor igual a $\lceil n/2 \rceil$.

Esse trabalho aborda o problema de encontrar o valor máximo do tamanho da maior subsequência crescente de uma permutação dos n primeiros números naturais quando tanto inversões quanto rotações da permutação são permitidas. Segundo nosso conhecimento, nenhuma fórmula para este valor é conhecida.

Na resolução deste problema, propõe-se uma abordagem de filtragem sobre o processamento exaustivo das permutações dos n primeiros naturais. Apesar dessa filtragem, o problema continua crescendo de forma exponencial em relação à n e se torna computacionalmente intratável na prática para valores pequenos de n. Isto motiva a utilização de técnicas de paralelismo na implementação do algoritmo proposto.

O uso de GPUs é cada vez mais comum para a resolução de problemas gerais da computação. O grande número de processadores que uma GPU oferece facilita a resolução de muitas aplicações que necessitam de um alto poder computacional. No entanto, a modelagem de problemas em GPUs é complexa. Muitos problemas são naturalmente paralelizáveis, enquanto outros não podem ser abordados de forma paralela. É preciso também conhecer o funcionamento de uma GPU, como ocorre a comunicação e seu processamento. Um programa paralelo mal estruturado pode ter um desempenho muito pior do que a sua versão sequencial. Assim, o programador precisa de conhecimentos específicos para conseguir melhorar o desempenho do algoritmo, explorando o paralelismo oferecido pela plataforma. O estudo de técnicas de paralelização e a arquitetura da placa gráfica utilizada são práticas necessárias para a capacitação do programador para desenvolver programas executáveis em GPUs.

Para a realização deste trabalho, foi desenvolvida inicialmente uma implementação sequencial do algoritmo. Em seguida, foi gerada uma versão paralela do algoritmo para a GPU. Neste trabalho, apenas foi paralelizada a geração de todas as permutações, por ser a parte da execução

que exige um volume maior de tempo de processamento. Para os algoritmos de extração de subsequências, como o LDS, não foi implementada a paralelização. Assim, esta parte foi executada sequencialmente na GPU.

A organização deste trabalho foi feita da seguinte forma. A seção 2 define formalmente o problema abordado. A seção 3 trata de aspectos relacionados com a plataforma CUDA e seu modelo de execução. A seção 4 descreve a forma em que as permutações são geradas. A seção 5 descreve os algoritmos utilizados e suas implementações. Na seção 6, explicitam-se os experimentos realizados e seus resultados. Por fim, as conclusões e trabalhos futuros são discutidos na seção 7.

2. O Problema da Máxima Subsequência Crescente Permitindo Inversões e Rotações

Sendo o problema atacado a determinação de um valor máximo computado dentre todas as permutações dos n primeiros números naturais, a sua dificuldade de resolução está no número exponencial de permutações em relação a n. Neste trabalho, foi criado um processo de filtragem para diminuir o número de permutações a serem processadas, sem modificar o resultado final. Vamos considerar n um número inteiro positivo maior ou igual a 1; S_n o conjunto de todas as permutações formada pelos n primeiros números naturais; π um elemento de S_n; R^i um operador que rotaciona uma permutação i unidades; I um operador que inverte uma permutação; $S_R(\pi)$ o conjunto gerado por $R^i\pi$, $0 \leq i \leq n-1$; $S_{RI}(\pi)$ o conjunto formado por elementos gerados por $R^i\pi \cup IR^i\pi$, $0 \leq i \leq n-1$.

Neste trabalho, deseja-se computar o maior valor possível sobre todos as permutações $\pi \in S_n$, para o tamanho mínimo da maior subsequência crescente existente em alguma permutação $\pi' \in S_{RI}(\pi)$. Em termos matemáticos pode ser representado por:

$$MAX_{\pi \in S_n}(MIN_{0 \leq i \leq n-1}(MIN(LIS(R^i\pi), LDS(R^i\pi)))$$

Observe que o cômputo de LDS é equivalente ao cômputo de LIS na permutação invertida.

Chamamos de $F(\pi)$ a função dentro do máximo, logo temos:

$$F(\pi) = MIN_{0 \leq i \leq n-1}(MIN(LIS(R^i\pi), LDS(R^i\pi))$$
$$MAX_{\pi \in S_n}(F(\pi))$$

Dado que o conjunto S_n pode ser extremamente grande, é importante fazer um esforço para diminuir a quantidade de elementos imprescindíveis de serem avaliados. Existem elementos de S_n diferentes que geram o mesmo resultado de F por processarem o mesmo conjunto devido às rotações e inversões. Dessa forma, só é necessário o cômputo de um deles. As sequências [1-2-3-4] e [2-3-4-1], por exemplo, são dois elementos de S_n diferentes que geram o mesmo conjunto S_{RI}, pois um pode ser gerado através do outro por rotações.

Toda permutação dos primeiros n naturais é rotação de alguma permutação que começa com o número 1, portanto, apenas estas precisam ser avaliadas. Ainda assim, todas as permutações que começam em 1 podem ser geradas por meio da inversão e rotação de uma outra permutação que também começa em 1. Para evitar a avaliação das duas permutações, consideramos apenas aquelas que, além de começar em 1, tem o 2 à frente do 3. Toda sequência que, começando em 1, tenha o 3 à frente do 2 pode ser invertida e depois rotacionada apenas uma vez para passar o 1 da última posição para a primeira. Dessa forma, o 2 fica à frente do 3.

3. Programação paralela em Unidades de Processamento Gráfico (GPU)

O interesse no uso de unidades de processamento gráfico (GPUs) como arquiteturas de computação de propósito geral vem crescendo nos últimos anos. GPUs apresentam uma quantidade massiva de processadores, tornando-as atrativas para a computação de alto desempenho. Por outro lado, a cache e o controle são pequenos quando comparados às CPUs modernas. O modelo de execução garante que enquanto threads estão bloqueadas esperando por dados, outras podem ser escalonadas para execução (dado um grau de ocupação suficiente). Atualmente,

vários frameworks facilitam o acesso dos programadores às capacidades de computação paralela da GPU. Neste documento nos referiremos especificamente ao *Compute Unified Device Architecture*, CUDA (NVIDIA 2015).

CUDA é uma arquitetura criada em 2006 pela NVIDIA que possibilita a execução de códigos em paralelo em GPUs da NVIDIA. Foi projetada para executar programas altamente escaláveis, possibilitando que os vários núcleos sejam utilizados ao mesmo tempo.

Nosso problema se trata de uma computação exaustiva, um mesmo fluxo sendo executado várias vezes com dados diferentes, e sem dependências entre eles. Computações embaraçosamente paralelas com alto paralelismo de dados, como esse caso, são ideais para serem executadas na GPU desde que os dados sejam suficientemente massivos. Abordagens baseadas em paralelismo de dados são adequadas para a GPU. Quando uma tarefa para enquanto aguarda por dados, a latência de memória pode ser escondida intercalando-se a execução de outras tarefas. Isso poderia ser dificultado em casos em que as tarefas estão fortemente ligadas, pois pode acontecer de não existir tarefas prontas para intercalação.

Para melhor compreender os algoritmos da GPU, é necessário o conhecimento de algumas definições de CUDA e da arquitetura da GPU.

O modelo de computação GPU + CPU é heterogêneo: a GPU atúa como co-processador da CPU para acelerar operações computacionalmente intensivas. Dessa forma, o programador precisa definir separadamente o que será executado na CPU e o que na GPU. As tarefas são submetidas pela CPU na GPU como chamadas a função com uma sintaxe específica que inicia a execução do código a ser executado pelas threads da GPU, chamado de *kernel*. O código é executado na GPU em grupos de threads que executam a mesma construção simultaneamente. Esses grupos são chamados de *warps*. Nas GPUs NVIDIA atuais, o tamanho de um warp é de 32 threads.

A CPU e a GPU não compartilham memória, por isso é necessário transferir explicitamente os dados entre as duas plataformas. A otimização das transferências CPU-GPU é fundamental para o desenvolvimento de programas paralelos eficientes na GPU (Gregg e Hazelwood 2011, Luong, Melab e Talbi 2013). O mapeamento de um subproblema para a thread correspondente sem intervenção da CPU é uma forma de minimizar essas transferências. Entretanto, no caso do presente trabalho, não é um procedimento trivial.

A principal prática utilizada em GPUs para melhorar o desempenho, é a maximização do *throughtput*. Isto é, maximizar a quantidade de tarefas executadas por unidade de tempo. Já na CPU, os esforços são centralizados na diminuição da latência, ou seja, diminuir o tempo gasto para o processamento de uma tarefa. Ambas abordagens foram aplicadas nesse trabalho com intuito de aumentar seu desempenho. O maior volume de computação é realizado na GPU, distribuída de maneira que as *threads* sempre fiquem ocupadas. Após o processamento na GPU, os dados são transferidos para a CPU, para realizar as últimas etapas do algoritmo. Nestas etapas, não é necessário um alto índice de processamento de dados, sendo ideal para a CPU.

CUDA organiza a computação paralela usando as abstrações de threads, blocos e grades. Blocos são grupos de threads. Grade é um grupo de blocos. A hierarquia de memória da arquitetura CUDA é composta por vários tipos de memória: local, compartilhada, constante, de textura e a global. Elas são otimizadas para diferentes padrões de acesso e diferem no tamanho máximo, a visibilidade e a persistência entre chamadas ao kernel. Por exemplo, o acesso à memória compartilhada é mais rápido que o acesso à memória global em duas ordens de magnitude, mas o tamanho dela é bem menor. Diferente da memória global, a memória compartilhada não persiste entre chamadas a kernel. Para a utilização desses vários tipos de memórias, é necessário uma declaração explícita na variável, ou então ela será considerada como uma variável local. Nesse trabalho, utilizamos somente variáveis locais e compartilhadas.

3.1. Configuração

A determinação do tamanho do bloco não é simples. Em geral, é preferível utilizar um tamanho grande de bloco, a fim de maximizar a ocupação do dispositivo. No entanto, existe um

compromisso entre o tamanho do bloco e o número de registros e a quantia de memória compartilhada disponíveis para as threads do bloco, uma vez que esses recursos são compartilhados. Também existe um limite de *threads* que podem ser lançadas por bloco de acordo com a quantidade de *cores*. A placa utilizada foi a GTX 690 que inclui duas GPUs. Isto permite processar metade das permutações em cada dispositivo. Cada uma dessas GPUs possui 48KB de memória compartilhada por bloco. Para o cálculo de quantas *threads* terá por bloco foi utilizado uma planilha disponibilizada no site da Nvidia que faz o cálculo da ocupação da GPU. Colocando as informações de gasto de memória compartilhada por bloco[Seção 5], a planilha produz um gráfico da ocupação da GPU pela quantidade de *threads* por bloco. Com esses resultados, escolhemos o número de 128 que produz uma ocupação de 100% da GPU. Em todos os códigos paralelos foram lançados 10240 *thread* por GPU. Esse valor decidido empiricamente, pois apresentava os melhores resultados.

4. Enumeração

Para melhorar o desempenho do programa paralelo foi criado um algoritmo de enumeração das permutações de interesse, como feito em (Luong et al. 2013). Dessa forma,(i) cada *thread* consegue processar independentemente uma parte do conjunto sem sobreposição de processamento, (ii) evitando a necessidade de gerar cada sequência na CPU e copiá-las para a GPU, com o consequente ganho em termos de diminuição do gargalo que constitui a transferência de dados. O mapeamento entre uma permutação e a *thread* da GPU correspondente não é trivial.

O algoritmo de enumeração é executado por cada *thread*, e recebe como parâmetro um número inteiro não-negativo (o identificador da thread mantido pelo runtime CUDA) e gera uma permutação dos primeiros n números naturais com o 1 na primeira posição e o 2 à frente do 3. Dentro da cardinalidade do conjunto de permutações de interesse, dois valores diferentes do parâmetro gerarão permutações diferentes.

Seja a função $menores(\pi, i)$ que retorna a quantidade de elementos da sequência que são menores do que o i-ésimo elemento e estão em uma posição superior a posição i Dado $\pi = [p_1, p_2, \ldots, p_n]$ o cálculo do índice da permutação é dado por:

$$indice = \sum_{i=1}^{n-1} menores(\pi, i) * (n - i)! \quad (1)$$

O somatório computa o número de permutações em ordem lexicográfica menores que π. A função $menores(\pi, i)$ retorna quantas modificações aconteceram nessa posição, enquanto (n-i)! representa quantas combinações são necessárias antes de movimentar o elemento π. Podemos ver na Tabela 1 um exemplo das sequências com n=3, e seus respectivos índices:

Tabela 1: Exemplo de sequência e seus índices para S_3

Sequência	índice
1-2-3	0
1-3-2	1
2-1-3	2
2-3-1	3
3-1-2	4
3-2-1	5

Em nossa aplicação, temos interesse na inversa desta função, isto é, dado um índice deseja-se determinar a permutação correspondente, como mostra o Algoritmo 1. Dado um índice, será modificada a posição de cada elemento da sequência de índice 0, indo da esquerda para a direita. A cada iteração é calculado o número de combinações possíveis com a parte ainda não processada da permutação, representado no algoritmo 1 pela variável fat. A divisão do índice por esse

valor representa quantas modificações ocorreram no elemento a ser calculado, $numMovimentos$, e o valor final da posição a ser computada é o elemento que esta à $numMovimentos$ posições do elemento que esta sendo computado. É feita uma rotação para a direita da posição que está sendo computada até mais $numMovimentos$ posições, depois o índice é subtraído pelo $numMovimentos$ vezes um peso, que é a quantidade de combinações possíveis com a subsequência dos elementos a partir da posição i, fatorial(N-i.1).

Algorithm 1 A partir de um índice, e um valor n, gera uma permutação dos n primeiros naturais

função: GERARSEQUÊNCIALEXICOGRAFICA(índice, n)
$vetorSaida = [1 - 2 - 3 - .. - n]$ //Menor sequência lexicográfica
$i = 0$
enquanto $i < n - 1$ **faça**
$fat = fatorial(n - i - 1)$

$numMovimentos = indice/fat$
se $numMovimentos > 0$ **então**
$temp = vetorSaida[i + numMovimentos]$
Rotaciona para direita os elementos da posição i até a posição (i+numMovimentos)
//Diminui o índice pela quantidade de movimentos vezes seu peso
$indice = indice - fatorial(N - i - 1) * numMovimentos$
fim se
$i = i + 1$
fim enquanto
devolve vetorSaida
fim função:

Na Tabela 2 segue um exemplo prático de como o algoritmo funciona.

Tabela 2: Exemplo de geração de sequência com N=6 e i=169_3

i	índice	vetorSaida	Observaćão
-	169	1-2-3-4-5-6	Valores iniciais
0	49	**2**-1-3-4-5-6	numMovimentos = 169/5!=1
1	1	2-**4**-1-3-5-6	numMovimentos = 1/4!=0
2	1	2-4-**1**-3-5-6	numMovimentos = 1/3!=0
3	1	2-4-1-**3**-5-6	numMovimentos = 1/2!=0
4	0	2-4-1-3-**6**-5	numMovimentos = 1/1!=1

A partir das técnicas de enumeração acima, foi desenvolvido um algoritmo para enumerar apenas as permutações com 1 na primeira posição e o 2 à frente do 3. O modo como é computada a posição dos elementos é igual, porém o 1 é fixo na primeira posição, e primeiramente é feito o cálculo da posição dos números 2 e 3. Com as posições desses três elementos já definidas, é calculado o resto da sequência com o índice que sobrou usando o Algoritmo 1, o retorno dessa função é inserido nas posições que não estão ocupadas por nenhum dos três elementos citados anteriormente. No Algoritmo 5.1 primeiramente calculamos a quantidade de movimentos que o elemento 2 ou 3 fazem, $numDeslocamento2e3$, isso se dá pela divisão do índice pela quantidade de combinações possíveis sem movimentar os três elementos, fatorial(tamVet-3). Caso o valor de $numDeslocamento2e3$ seja maior do que zero, isso significa que a posição de 2 e/ou 3 não é a inicial. Agora é necessário fazer um laço para calcular a real posição do elemento 2 e do 3, sendo o segundo relativo ao primerio. Para que o elemento 2 se desloque da posição i para a $i + 1$, é necessário que o valor de $numDeslocamento2e3$ seja maior do que a quantidade de posições após

i, sem contar os elementos 1 e 2. O valor inicial da posição do elemento 2 é 1, $pos_{num2} = 1$. Enquanto $numDeslocamento2e3$ for maior do que zero, é feita a verificação se o elemento 2 pode ir para a próxima posição. Caso a afirmação seja verdadeira, então pos_{num2} é incrementado, e $numDeslocament2e3$ é decrementado igualmente pelo peso $tamVet - 2 - i$. Se ela for falsa, então significa que o elemento 2 está na posição 2, e o valor restante em $numDeslocamento2e3$ é a quantidade de movimentos restante para o elemento 3, ou seja, $pos_{num3} = posnum2 + 1 + numDeslocamento2e3$. Com as posições finais dos elementos 1,2 e 3 já calculadas, agora é gerado o restante dos elementos em ordem lexicográfica. O resto da divisão $index/fatorial(tamVet-3)$ representa o índice que será utilizado no Algoritmo1, e o resultado da função será inserido nas posições em branco do vetor final.

Algorithm 2 A partir de um índice gera uma permutação em que o elemento 2 está sempre na frente do elemento 3, e o primeiro elemento é o 1

função: GERASEQUENCIA(tamVetor, indice)
  $numDeslocamento2e3 = index/fatorial(tamVet - 3)$
  $pos_{num2} = 1$ //Posição padrão do elemento 2
  $i = 0$
  //Calcular a posição final do 2 e do 3. Como a quantidade de movimentos de 3
  //necessários para movimentar 2, depende da posição de 2 temos que fazer esse ciclo
  enquanto $numDeslocamento2e3 > 0$ **faça**
    //Caso o numero de deslocamento for maior que o necessário para movimentar
    //o elemento 2, incremente, se não calcule a posição de 3, e saia do laço.
    se $numDeslocamento2e3 >= (tamVet - 2 - i)$ **então**
      $pos_{num2} + + = pos_{num2} + 1$
      $numDeslocamento2e3 = numDeslocamento2e3 - (tamVet - 2 - i)$
    senão
      $pos_{num3} = pos_{num2} + 1 + numDeslocamento2e3$
      sair enquanto
    fim se
    $i = i + 1$
  fim enquanto
  $vetorSaida[0] = 1$
  $vetorSaida[pos_{num2}] = 2$
  $vetorSaida[pos_{num3}] = 3$
  $indiceResto = index\%fatorial(tamVet - 3)$
  $seqLexicografica = GeraSequenciaLexicografica(indiceResto, tamVet - 3)$
  Insira nas posições vazias do vetorSaida os elementos de seqLexicografica
devolve vetorSaida
fim função:

5. Algoritmos utilizados

Para a determinação da melhor distribuição de trabalho entre a GPU e a CPU em termos de tempo de execução foram criados três códigos diferentes:

- O primeiro (Serial) é o puramente serial e executa na CPU.
- No segundo (Serial-Paralelo), a CPU somente gera as permutações e as copia na GPU. Então a GPU faz o processamento das permutações.
- Já no último (Paralelo), tanto a geração das sequências quanto o processamento são feitos na GPU, mostrando a potência que tal tipo de processador pode ter para aplicações gerais.

5.1. Serial

No algoritmo Serial inicialmente é calculada a quantidade de elementos π de S_{RI}, que pode ser representada através da equação $fatorial(n-1)/2$. Com isso é possível gerar todos os elementos do conjunto utilizando o algoritmo . Para cada elemento π gerado, é feito o cálculo de $F(\pi)$ e armazenado o maior valor encontrado. Caso em uma das rotações ou inverções (LDS) de π for encontrado um valor menor ou igual ao maior valor já encontrado por um outro $F(\pi)$, então interrompemos o processamento do atual $F(\pi)$ e passamos para o próximo elemento.

5.2. Serial-Paralelo

Neste código, as permutações são geradas na CPU e armazenadas em uma matriz. Essa matriz é copiada para a GPU. Cada *thread* da GPU faz o cálculo do $MIN_{0\leq i\leq N-1}(MIN(LIS(R^i\pi),$ $LDS(R^i\pi))$ Em seguida, se armazena o máximo entre o resultado, e o valor que já estava armazenado. Depois que todas as sequências de S_{RI} já foram geradas e produzidas, fazemos uma redução dos resultados obtidos pela GPU. Esta redução foi feita na CPU, pois o tempo gasto pela mesma para criar as sequências sem deixar o processador gráfico muito tempo desocupado é muito baixo. Em CUDA, a CPU e a GPU podem trabalhar simultaneamente. Em consequência, mesmo que um *kernel* esteja executando na GPU, a CPU irá continuar a criar sequências. O principal objetivo deste código é mostrar o impacto da transferência de memória entre os processadores no desempenho do sistema.

5.3. Paralelo

O algoritmo 3 é a parte executada pela GPU. Onde cada *thread* da GPU executará totalmente independente uma da outra. Para que isso aconteça, foi necessário gerar as sequências utilizando o Algoritmo 5.1. Com isso cada *thread* consegue saber as sequências que vai processar a partir de seu índice. Cada *thread* possui uma posição específica do vetorResultado, onde será armazenado o maior valor de $F(\pi)$ encontrado. O Algoritmo 4 é a parte serial do programa que será executada pela CPU. Ele simplismente aloca o vetorResultados na GPU, pois é a única variável que precisa ser transferida de volta para a CPU, para que possa realizar a redução do vetorResultado O processador escolhido para realizar a redução não interfere significamente nos resultados, pois o tempo gasto para reduzir 10240 elementos é insignificante quando comparado ao tempo gasto com o resto do processamento.

Algorithm 3 Código Paralelo a executar na GPU

1: **função:** KERNEL:DECIDELS(vetorResultado, tamVetor, numThreadInicial)
2: $threadIdGlobal = numThreadInicial + threadId$
3: $indice = threadIdGlobal$
4: $indiceMax = fatorial(tamVetor-1)/2$
5: **enquanto faça** $indice < indiceMax$
6: $sequencia = GeraSequencia(indice, tamVetor)$
7: $LSminP_n = \infty$
8: **para** i $\leftarrow$ 0 **até** tamVetor **faça**
9: $lLIS = LIS(sequencia[threadId], tamVetor)$
10: $LSminP_n = MIN(lLIS, LSminP_n)$
11: $lLDS = LDS(sequencia, tamVetor)$
12: $LSminP_n = MIN(lLDS, LSminP_n)$
13: Rotaciona a sequência por 1 elemento
14: **fim para**
15: $indice = indice + numThreadsTotal$
16: $vetorResultado[threadIdGlobal] = MAX(LSminP_n, vetorResultado[threadIdGlobal])$
17: **fim enquanto**
18: **fim função:**

Algorithm 4 Sendo S o conjunto de todos as sequências de N elementos, que não podem ser geradas com permutações e inversões entre si, e R elemento de S. Essa função calcula o maior LIS e LDS do conjunto.

```
1: função: LSSERIALPARALELO(tamVetor, numThreadsPorGPU, numGPUs)
2:     Aloca memória na CPU e na GPU para a variável: vetorResultado[numThreadsTotal]
3:     Configura o número de GPUs que vão ser utilizadas
4:     para i ← 0 até numGPUs faça
5:         Lança kernel: decideLS(vetorResultado, tamVetor, i*numThreadsPorGPU)
6:     fim para
7:     Transfere dados do vetorResultado para a CPU
8:     LSmaxS = Redução com operador de máximo do vetorResultado devolve LSmaxS
9: fim função:
```

6. Experimentos e Resultados

Os experimentos foram realizados em um computador Intel(R) Core(TM) i7-980 com 12 MB de cache e 3.33 GHz e 24 GB de RAM e uma placa NVIDIA GeForce GTX 690 (dupla) com 1536 cores por GPU.

O código paralelo foi submetido à GPU para execução em 10240 *threads*. A configuração escolhida foi 128 threads por bloco e 80 blocos por grid. O valor foi escolhido empiricamente para diminuir o tempo de execução.

Primeiramente, os algoritmos apresentados na seção 5 foram executados variando o n com um tempo limite de 2 dias. Os resultados obtidos são mostrados na tabela 3 e no gráfico da figura 1.

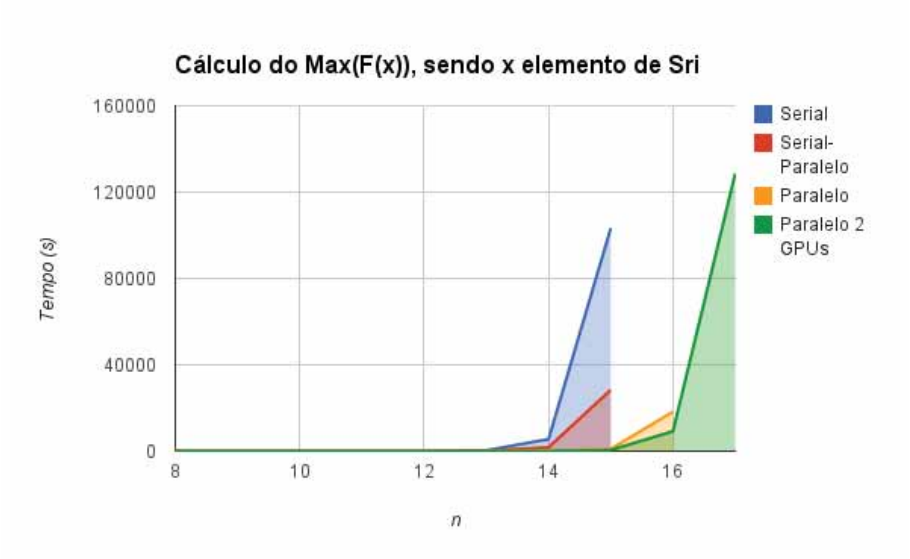
A placa de vídeo Geforce GTX 690 inclui duas GPUs. É possível tirar proveito disso através da divisão do conjunto de permutações entre as duas GPUs. O problema tratado tem a vantagem de ser altamente paralelizável de forma que não é necessária a comunicação entre os dois dispositivos, o que nessa placa ainda é uma operação onerosa. Os resultados da execução do algoritmo paralelo em duas GPUs simultaneamente são identificados na tabela como *Paralelo com 2 GPUs*.

Tabela 3: Resultados dos algoritmos apresentados na sessão 5. Valores em segundos

n	Resultado	Serial	Serial-Paralelo	Paralelo	Paralelo com 2 GPUs
7	3	0.000312	0.000824	0.00041	0.000425
8	3	0.004623	0.002194	0.00043	0.000442
9	3	0.036178	0.007122	0.00101	0.000746
10	4	0.203792	0.054921	0.00753	0.004347
11	4	2.521092	0.654392	0.04712	0.033138
12	4	37.589523	8.140342	0.48462	0.251217
13	5	351.769876	115.336479	8.14167	4.164411
14	5	5571.729102	1766.363525	53.21826	28.527290
15	5	103319.906250	28378.007812	983.17404	493.864563
16	6	-	-	18313.17268	9207.769531
17	6	-	-	-	128550.689126

A utilização da paralelização na GPU permitiu encontrar valores do máximo de $F(\pi)$ para $n = 17$, o que não era possível devido ao crescimento exponencial do número de sequências do conjunto S_n. Os valores encontrados dão suporte à conjectura de que a solução para o problema abordado é $\lceil n/3 \rceil$. A paralelização na GPU só vale a pena para grandes quantidades de dados, sendo pior que o serial para $n < 7$. Nestes casos, o custo de transferência de dados para a GPU sobrepassa

Figura 1: Resultados dos algoritmos apresentados na sessão 5. Valores em segundos



o custo do cômputo. Ao comparar o resultado do algoritmo puramente paralelo para $n = 7$ e $n = 8$, podemos ver que não foi utilizado todo o potencial da GPU, já que o tempo gasto no cálculo foi praticamente o mesmo, independentemente de o primeiro ter 7 vezes mais elementos que o segundo. Com a utilização da GPU, conseguimos no melhor caso um programa aproximadamente 200 vezes mais rápido que o serial, Figura 2.

Tabela 4: Speedup dos algoritmos em paralelo

n	Serial-Paralelo	Paralelo	Paralelo com 2 GPUs
8	2.11	10.75	10.45927602
9	5.08	35.82	48.49597855
10	3.71	27.06	46.8810674
11	3.85	53.50	76.07858048
12	4.62	77.56	149.6296946
13	3.05	43.21	84.47049919
14	3.15	104.70	195.3122467
15	3.64	105.09	209.2069648

Com o objetivo de explicar porque o algoritmo Serial-Paralelo fica muito pior que o algoritmo Paralelo, foram coletados os dados referentes ao tempo gasto para fazer as operações de criar a sequência, transferir os dados da CPU para a GPU, e calcular a função F para as permutações produzidas, para um conjunto de 10240 permutações de S_{RI}, esse número foi escolhido porque é o mínimo necessário para lançar todas as *thread* igual é feito no algoritmo Paralelo. Como pode ser visto claramente na figura 3, o tempo gasto para transferir os dados é muito maior que o tempo para criar e processar as permutações somadas. Mesmo modificando o tamanho da sequência, não interfere significativamente na razão entre o tempo de transferência e as demais operações. Em contrapardida, o algoritmo Paralelo não necessita transmitir quase nenhuma informação, pois tudo é calculado dentro da GPU.

Figura 2: Gráfico dos Speedups

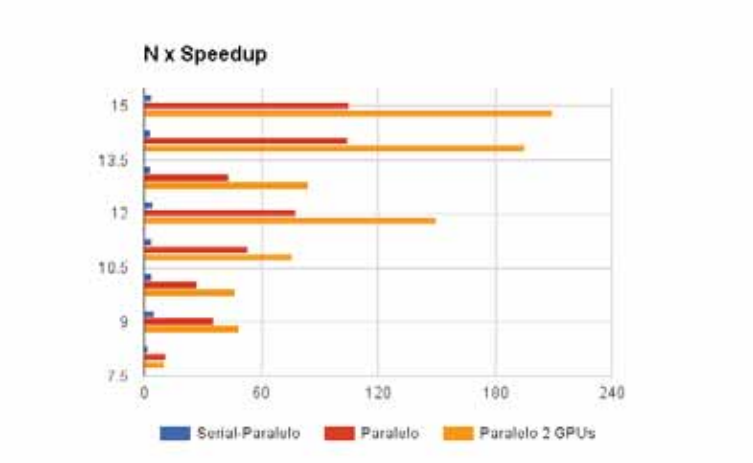
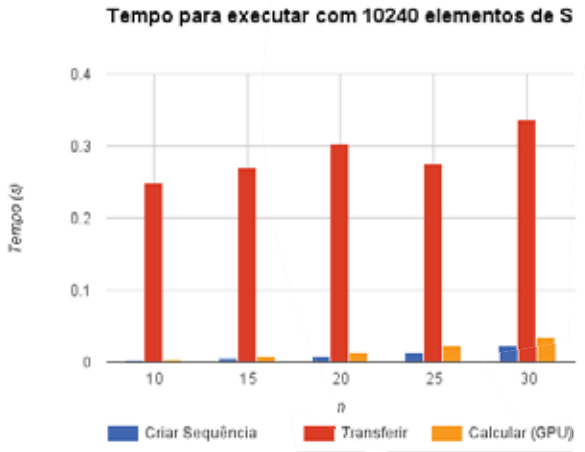


Tabela 5: Resultados dos algoritmos mencionados na sessão anterior. Valores em segundos

N	Criar Permutação (Serial)	Transferir	Calcular (GPU)	
10	0.002851	0.248654	0.004055	
15	0.005856	0.269738	0.008437	
20	0.009135	0.304349	0.014233	
25	0.013275	0.276138	0.022354	
30	0.022260	0.336750	0.035502	

Figura 3: Grafico que mostra o tempo para calcular $F(\pi)$ em funcao do n



7. Conclusões e Trabalhos Futuros

Esse artigo foca na resolução, por meio de algoritmos paralelos, do cômputo do máximo de F(s) sendo s um elemento do conjunto S_{RI}. Acredita-se que este valor seja sempre igual a $\lceil n/3 \rceil$.

Com os resultados obtidos nos experimentos realizados neste trabalho, foi verificado que a hipótese acima é verdadeira para os valores de n que foram possíveis de processar dentro do tempo limite. Isso foi melhor observado com a utilização da implementação paralela do algoritmo devido à possibilidade de obter resultados para valores mais elevados de n. A abordagem que alcançou melhor resultado foi aquela que utilizou da enumeração do conjunto S_{RI}. Esta possibilitou a criação da permutação concorrentemente dentro da GPU resultando em uma diminuição do tempo de execução em até 200 vezes em relação ao código serial. É válido frisar que foram obtidos resultados para $n = 17$ na versão paralela e apenas para $n = 15$ na serial.

Possíveis trabalhos futuros poderião proporcionar melhora do algoritmo paralelo, como a paralelização do LIS e do LDS, e desenvolvimento de novas formas de reutilizar cálculos já realizados em sequências diferentes. Essas modificações permitiriam encontrar o resultado de valores mais elevados de n do que os encontrados durante a realização desse trabalho.

Agradecimentos

Agradeço ao CEFET-MG por me proporcionar um crescimento acadêmico, e por financiar a publicação do artigo. Este trabalho também foi parcialmente financiado pelo CNPq e Fapemig.

Referências

Casazza, M., Ceselli, A. e Nunkesser, M. (2012), 'Efficient Algorithms for the Double Traveling Salesman Problem with Multiple Stacks', *Comput. Oper. Res.* **39**(5), 1044–1053.

Golumbic, M. C. (2004), *Algorithmic Graph Theory and Perfect Graphs (Annals of Discrete Mathematics, Vol 57)*, 2nd edn, North-Holland Publishing Co., Amsterdam, The Netherlands, The Netherlands.

Gregg, C. e Hazelwood, K. (2011), Where is the data? Why you cannot debate CPU vs. GPU performance without the answer, *in* 'Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software', ISPASS '11, IEEE Computer Society, pp. 134–144.

Luong, T. V., Melab, N. e Talbi, E.-G. (2013), 'GPU Computing for Parallel Local Search Metaheuristic Algorithms', *IEEE Trans. Computers* **62**(1), 173–185.

NVIDIA (2015), 'CUDA programming guide 7.0'. Acessado em 01/05/2015.

Petersen, H. L. e Madsen, O. B. G. (2009), 'The Double Travelling Salesman Problem with Multiple Stacks - Formulation and heuristic solution approaches', *European Journal of Operational Research* **198**(1), 139–147.

Romik, D. (2015), *The Surprising Mathematics of Longest Increasing Subsequences*, Cambridge University Press.