Programação dinâmica paralela em *GPU* para os problemas da mochila uni e bi-dimensional

Dayllon V. X. Lemos[1], **Humberto J. Longo**[1], **Wellington S. Martins**[1]

[1]Instituto de Informática (INF)
Universidade Federal de Goiás (UFG)
74960-970 – Goiânia – GO – Brasil

dayllonxavier@discente.ufg.br,{longo,wsmartins}@ufg.br

Abstract. *The work addresses the Multidimensional Knapsack Problem. Parallel algorithms on* GPU*, based on dynamic programming, are presented for the sub cases with one and two dimensions. An algorithm for the joint resolution of several instances of the problem is also presented. Computational tests performed with implementations of these algorithms showed a considerable gain in efficiency, when compared to their sequential versions.*

Keywords. *Multidimensional Knapsack, Graphics Processing Unit,* MKP*,* GPU*, Dynamic Programming.*

Resumo. *O trabalho apresenta algoritmos paralelos de granularidade fina, baseados em programação dinâmica, para a resolução do Problema da Mochila Uni e Bi-Multidimensional. É apresentado também um algoritmo para a resolução conjunta de várias instâncias do problema. Os testes computacionais realizados com implementações dos algoritmos em* GPU *mostraram um ganho de eficiência considerável, em comparação às suas versões sequenciais.*

Palavras-chave. *Mochila Multidimensional, Unidade Gráfica de Processamento,* MKP*,* GPU*, Programação Dinâmica.*

1. Introdução

O Problema da Mochila Multidimensional 0–1 (*0–1 Multidimensional Knapsack Problem, MKP*) pertence à classe dos problemas $\mathcal{NP}$-difíceis [Kellerer et al. 2004]. O objetivo no *MKP*, dado um conjunto de objetos associados a valores (ganho) e a pesos (custo), é selecionar um subconjunto destes objetos, maximizando-se o ganho total e respeitando-se as limitações de capacidade da mochila em cada uma de suas dimensões.

O *MKP* possui diversas aplicações em áreas como otimização combinatória, criptografia, problemas de logística e tomada de decisão, dentre outras. São encontrados na literatura relatos de aplicações em atividades tão diversas como: corte de materiais [Gilmore and Gomory 1966], orçamento de capital e alocação de recursos [Lorie and Savage 1955], planejamento de transporte de cargas [Bellman 1957, Shih 1979], alocação de processadores e bancos de dados em grandes sistemas distribuídos [Gavish and Pirkul 1982] e desenvolvimento de estratégias para controle e prevenção de poluição [Bansal and Deep 2012], entre outras.

As unidades de processamento gráfico (*Graphics Processing Unit – GPU*s) são muito conhecidas e aplicadas na área de desenvolvimento de jogos, para renderização

gráfica e processamento de imagens. As *GPU*s também permitem realizar processamento paralelo em aplicações de propósito geral. Essa técnica é conhecida como *General Purpose Graphics Processing Unit (GP–GPU)* e consiste na utilização de *GPU*s para resolver problemas que não envolvam aspectos gráficos. Assim, além de poderem ser utilizadas como um meio eficiente no processamento de grandes volumes de dados, podem mostrar-se eficazes na resolução de instâncias de grande porte de problemas de otimização, tal como o *MKP*. Por exemplo, quando o algoritmo usado requer a resolução repetitiva de algum subproblema em particular e essa tarefa pode ser executada em paralelo em *GPU*s.

Neste trabalho são apresentados algoritmos paralelos de granularidade fina, baseados na técnica de programação dinâmica, para os casos particulares do *MKP* com uma e duas dimensões. Embora esses casos tenham uma estrutura limitada, ambos ainda pertencem à classe de problemas $\mathcal{NP}$-difíceis.

A Seção 2 apresenta a notação utilizada no restante do artigo, além de definir formalmente os casos particulares abordados. A Subseção 3.1 discorre sobre a resolução sequencial do *KP1* e do *KP2*. A Subseção 3.2 explicita a transformação dos algoritmos sequenciais de resolução do *KP1* e *KP2* para versões paralelas de granularidade fina. A Subseção 3.3 apresenta técnicas para a resolução de múltiplas instâncias paralelas do *KP1* e do *KP2*. A Seção 4 é dedicada aos resultados computacionais obtidos em testes com cada um dos algoritmos descritos na Seção 3. Por fim, na Seção 5, são apresentadas as considerações finais e algumas ideias para trabalhos futuros.

2. Definições formais

O Problema da Mochila Multidimensional (*Multidimensional Knapsack Problem, MKP*) é definido pelos seguintes elementos: capacidades c_k, $1 \leqslant k \leqslant m$, das m dimensões da mochila, valores (lucros – *profits*) p_i, $i \leqslant i \leqslant n$, associados aos n objetos e pesos (*weight*) $w_{i,k}$ dos objetos i, $1 \leqslant i \leqslant n$, nas dimensões k, $1 \leqslant k \leqslant m$. O objetivo no *MKP* é selecionar um subconjunto dos n objetos, de forma que o lucro total seja maximizado e as capacidades c_k, $1 \leqslant k \leqslant m$, das m dimensões da mochila não sejam ultrapassadas. O *Problema da Mochila Unitária* (*Knapsack Problem – KP1)* e o *Problema da Mochila Bidimensional* (*2-Dimensional Knapsack Problem – KP2)* são os casos particulares do *MKP* em que $m = 1$ e $m = 2$, respectivamente.

Com relação ao *MKP*, seja $z_j(c_1, \ldots, c_m)$ o valor máximo que pode ser obtido considerando-se apenas os primeiros j objetos, para algum $1 \leqslant j \leqslant n$, e as capacidades $c_1, \ldots, c_m$, das m dimensões da mochila. Portanto, $z_j(c_1, \ldots, c_m)$ pode ser calculado através da seguinte relação de recorrência:

$$z_j(c_1, \ldots, c_m) = \begin{cases} 0, & \text{se } j = 0; \\ z_{j-1}(c_1, \ldots, c_m), & \text{se } c_k < w_{j,k}; \\ \max \left\{ \begin{array}{l} z_{j-1}(c_1, \ldots, c_m), \\ z_{j-1}(c_1 - w_{j,1}, \ldots, c_m - w_{j,m}) + p_j \end{array} \right\}, & \text{se } c_k \geqslant w_{j,k}. \end{cases} \quad (1)$$

A recorrência (1) tem como caso base $j = 0$, isto é, quando nenhum objeto é considerado, o valor máximo obtido, trivialmente, é 0. O segundo caso da recorrência baseia-se no fato de que algum peso $w_{j,k}$ ser superior à capacidade c_k da mochila. Neste caso, o objeto j não pode ser adicionado ao subconjunto solução, o qual só poderá conter objetos com índices

menores do que j. O último caso da recorrência é quando, para todo k, $1 \leqslant k \leqslant m$, $w_{j,k} \leqslant c_k$ e, portanto, a adição do objeto j não extrapola nenhuma capacidade da mochila. Assim, o valor de $z_j(c_1, \ldots, c_m)$ será o máximo entre não considerar e considerar o objeto j como elemento pertencente ao subconjunto solução.

O valor ótimo da recorrência (1) pode ser obtido por programação dinâmica, usando-se uma estratégia *top–down* combinada com a técnica de *memoização* [Pfeffer 2007]. Contudo, a abordagem considerada neste artigo foi a *bottom–up*, devido à sua facilidade de paralelização, como descrito na Subseção 3.2. Esta abordagem caracteriza-se por primeiro resolver subproblemas triviais e usar as soluções de subproblemas já computados na resolução de subproblemas mais complexos, até que a solução do problema original esteja computada (*etapa de combinação* de soluções).

3. Abordagens de resolução do *KP1* e do *KP2*

Nos algoritmos apresentados nesta seção, a notação $\&$ representa a operação lógica *AND* aplicada *bit* a *bit* a dois valores inteiros. Assim, $i \,\&\, 1 = 0$ se i é um número par ou 1 caso contrário. Essa expressão é usada para se alternar entre a primeira e a segunda linha de matrizes de estados de 2 linhas e várias colunas, definidas nos vários algoritmos. A notação $A[m][n] \leftarrow \{0\}$, além de definir A como uma matriz de dimensão $m \times n$, significa que todos as posições de A são inicializadas com o valor 0 (notação similar também é usada para matrizes 1- e 2-dimensionais).

Os algoritmos paralelos de granularidade fina descritos neste artigo utilizam os princípios da *API CUDA* (vide, por exemplo, [Garland et al. 2008, Nickolls et al. 2008] para mais detalhes sobre *GPU* e *CUDA*). Em *CUDA* os dados acessados pelo *kernel* (procedimento (código) que será executado em paralelo na *GPU*) devem se encontrar na memória da *GPU* e não na memória principal da *CPU*. Assim, antes da execução de um *kernel*, os dados a serem utilizados por este, devem ser copiados para a memória da *GPU*. Também, após a execução do *kernel*, os dados de interesse devem ser copiados de volta para a memória principal. Um programa, em processamento na *CPU*, pode chamar uma rotina de *kernel*, especificando a quantidade de blocos da *GPU* que serão utilizados e quantas *threads* por bloco serão lançadas. O sufixo $_{GPU}$ foi usado para explicitar as estruturas existentes apenas na memória da *GPU*. Nesse caso, a sua referencia é um ponteiro (endereço) para a estrutura na memória da *GPU* e uma declaração como $A \leftarrow A_{GPU}$ indica cópia de dados da memória da *GPU* para a memória da *CPU*.

Os parâmetros $qtdeB$ e $qtdeT$ dos algoritmos paralelos descritos nesta Seção representam, respectivamente, a quantidade de blocos e a quantidade de *threads* por bloco lançadas na *GPU*. Algumas funções são invocadas para simplificar a descrição dos algoritmos paralelos: $obtemQtdeBlocos()$, retorna o valor de $qtdeB$; $obtemQtdeThreadsBloco()$ retorna o valor de $qtdeT$; $obtemThreadId()$ retorna o identificador tId da *thread* dentro do bloco, $0 \leqslant tId < numT$; $obtemBlocoId()$ retorna o identificador bId do bloco, $0 \leqslant bId() < qtdeB$.

3.1. Algoritmo sequencial

A resolução sequencial do *KP2*, usando-se a técnica de programação dinâmica com estratégia *bottom-up*, a partir da recorrência 1, é apresentada no Algoritmo 1, o qual tem complexidade assintótica de tempo e espaço $\mathcal{O}(n \cdot c_1 \cdot c_2)$. Um algoritmo de resolução

sequencial do *KP1* pode ser facilmente obtido desse algoritmo, adaptando-se as estruturas e processos para considerar apenas uma dimensão.

No Algoritmo 1 a matriz M armazena as soluções dos subproblemas da instância original. Considere que os n objetos da instância tenham índices $1, 2, \ldots, n$ e seja um certo índice $n' \leqslant n$. Na etapa de combinação do algoritmo são necessárias apenas as soluções dos subproblemas com algum subconjunto dos $(n' - 1)$ primeiros objetos para a composição das soluções dos subproblemas considerando os n' primeiros objetos, como é evidente pela recorrência (1). Após isso, é necessário recuperar recursivamente os objetos que foram efetivamente usados na composição da solução final (etapa de *backtracking*). Assim, não é necessário manter M com dimensão $n \times (c_1 + 1) \times (c_2 + 1)$, mas apenas com dimensão $2 \times (c_1 + 1) \times (c_2 + 1)$, onde cada posição $M[n' \,\&\, 1][c'_1][c'_2]$ guarda o valor da solução do subproblema considerando os objetos de índices menores ou iguais a n' e capacidades c'_1 e c'_2 ($0 \leqslant n' \leqslant n$, $0 \leqslant c'_1 \leqslant c_1$ e $0 \leqslant c'_2 \leqslant c_2$). Dessa forma, a cada iteração do algoritmo, pode-se alternar entre as duas únicas linhas de M para o armazenamento das soluções dos subproblemas com qualquer quantidade $q \geqslant 0$ de objetos, ou seja, linha 0 para q ímpar e linha 1 para q par.

Algoritmo 1: sequentialKP2Solver $(n, c_1, c_2, \boldsymbol{p}, \boldsymbol{w})$

Input: Número de objetos n; capacidades c_1 e c_2 das dimensões da mochila; vetor $\boldsymbol{p}$ de lucros; matriz $\boldsymbol{w}$ de pesos.

Output: Valor máximo z. Vetor *Opt* de *bits* que indica os objetos selecionados.

```
1  M[2][c1 + 1][c2 + 1] ← {0};      // Matriz de estados (2 × (c1 + 1) × (c2 + 1)).
2  U[n][c1 + 1][c2 + 1] ← {0};      // Matriz de bits (n × (c1 + 1) × (c2 + 1)).
3  para i ← 1 até n faça
4      para j1 ← 0 até c1 faça
5          para j2 ← 0 até c2 faça
6              M[i & 1][j1][j2] ← M[(i − 1) & 1][j1][j2];
7              se (j1 ⩾ w_{i,1} e j2 ⩾ w_{i,2}) então
8                  c'1 ← j1 − w_{i,1};
9                  c'2 ← j2 − w_{i,2};
10                 se (M[i & 1][j1][j2] < M[(i − 1) & 1][c'1][c'2] + p_i) então
11                     M[i & 1][j1][j2] ← M[(i − 1) & 1][c'1][c'2] + p_i;
12                     U[i][j1][j2] ← 1;
13 z ← M[n & 1][c1][c2];
14 Opt[n] ← {0};                    // Vetor de bits (n posições).
15 para i ← n até 1 faça
16     se (U[i][c1][c2] = 1) então
17         c1 ← c1 − w_{i,1};
18         c2 ← c2 − w_{i,2};
19         Opt[i] ← 1;
20 retorna z, Opt.
```

Contudo, essa representação da matriz M impede a recuperação dos objetos que fazem parte do subconjunto solução ótimo (na fase de *backtracking*). Para contornar tal situação, optou-se por utilizar uma segunda matriz U de elementos binários (*bits*), com

dimensão $n\times(c_1+1)\times(c_2+1)$, a qual permite economia no uso da memória (gestão otimizada da memória) disponível. Inicialmente, no passo 2, todos os objetos são marcados como não pertencentes ao subconjunto solução ($U[i][j_1][j_2]\leftarrow\{0\}$). Ao final do processamento do bloco de passos 3–12, faz-se $U[i][j_1][j_2]\leftarrow 1$, se o objeto i é adicionado ao subconjunto solução considerando-se os i primeiros objetos e capacidades j_1 e j_2 na primeira e segunda dimensão da mochila, respectivamente.

Na computação da matriz de estados M, inicialmente, considera-se que o objeto i não faz parte do subconjunto solução (passo 6). Em seguida, caso a adição do objeto i não ultrapasse nenhuma das duas capacidades da mochila (passo 7), é avaliado se sua adição representa ganho no valor máximo obtido com os $i-1$ primeiros objetos (passo 10). Em caso positivo, o lucro do objeto i é adicionado ao valor máximo obtido até então (passo 11). Após computar todos os estados da matriz M, o valor máximo da solução obtida para a instância original do problema encontra-se na posição $M[n][c_1][c_2]$ (passo 13). No caso particular em que $i=1$, há subproblemas sem nenhum objeto, Nesses casos, a solução é trivialmente 0 e definida pela inicialização da matriz M no passo 1 ($M[2][c_1+1][c_2+1]\leftarrow\{0\}$).

A matriz unidimensional Opt, com n *bits*, é usada na fase de *backtracking* (passos 15 à 19) para indicar quais objetos pertencem ou não ao subconjunto solução ótimo. Novamente, todos os objetos são inicialmente marcados como não pertencentes ao subconjunto solução ($Opt[n]\leftarrow\{0\}$, no passo 14). Note-se que a variável de controle do laço do passo 15 é decrementada de n até 1. Assim, se $U[i][c_1][c_2]=1$ (passo 16), o objeto i pertence ao conjunto solução ótimo (passo 19) e seus pesos são adequadamente decrementados das capacidades da mochila (passos 17–18). Após isso, o processo se repete para o novo valor de i e as capacidades remanescentes c_1 e c_2. Ao final, o valor z e o vetor Opt são retornados (passo 20) como solução do Algoritmo 1.

3.2. Algoritmos paralelos para o *KP1* e o *KP2*

Uma proposta para resolução paralela do *KP2*, implementada em *GPU*, é apresentada nos Algoritmos 2 e 3. O primeiro deve ser executado na *CPU* e faz a chamada ao segundo (*kernel*), o qual é executado em paralelo na *GPU*. O Algoritmo 2 é muito parecido ao Algoritmo 1, com as matrizes M_{GPU}, U_{GPU} e Opt equivalentes às matrizes M, U e Opt, respectivamente, sendo utilizadas com as mesmas finalidades. A variável z (passo 5) armazena o valor máximo da solução para a instância do problema e os passos de 8 à 12 realizam a fase de *backtracking* da programação dinâmica. Algumas adaptações nesses dois algoritmos, similares àquelas descritas na Seção 3.1, permitem a resolução paralela também do *KP1*. Nos passos 3–4 são realizadas as chamadas ao *kernel* que realiza, em paralelo na *GPU*, a computação da linha i da matriz de estados M_{GPU} (tarefa equivalente aos passos 4–12 do Algoritmo 1).

No Algoritmo 3 cada *thread* recebe um identificador tId, $0\leqslant tId<qtdeB\cdot qtdeT$ (passo 1) e uma variável numérica $passo$, com a quantidade de *threads* lançadas na *GPU*, é definida. Após isso, na ℓ-ésima iteração do laço contido nos passos 3–12, a *thread* de índice *tId* calcula o estado $M_{GPU}[i][tId+\ell\cdot passo]$. A iteração continua enquanto $tId+\ell\cdot passo<(c_1+1)\cdot(c_2+1)$. Dessa forma, todos os estados da linha i da matriz M_{GPU} são computados, com cada *thread* calculando aproximadamente $\frac{(c_1+1)\cdot(c_2+1)}{qtdeB\cdot qtdeT}$ estados.

A matriz bidimensional M_{GPU} ($2\times((c_1+1)\cdot(c_2+1))$) do Algoritmo 2, exerce a

mesma função que a matriz tridimensional M ($2\times(c_1+1)\times(c_2+1)$) do Algoritmo 1. Assim, o estado $M_{GPU}[i \,\&\, 1][c'_1 \cdot (c_2+1) + c'_2]$ é equivalente ao estado $M[i \,\&\, 1][c'_1][c'_2]$. A decisão de se manter a matriz de estados M_{GPU} com apenas 2 dimensões foi realizada para facilitar a escrita do *kernel*.

Algoritmo 2: parallelKP2Solver ($n, c_1, c_2, \boldsymbol{p}, \boldsymbol{w}, qtdeB, qtdeT$)

Entrada: Quantidade n de objetos; capacidades c_1 e c_2 das dimensões da mochila; vetor $\boldsymbol{p}$ de lucros; matriz $\boldsymbol{w}$ de pesos; quantidade $qtdeB$ de blocos usados na *GPU*; quantidade $qtdeT$ de *threads* lançadas em cada bloco.

Saída: Valor máximo z. Vetor Opt de *bits* que indica os objetos selecionados.

```
// Matriz de 2 × ((c1 + 1) · (c2 + 1)) estados iniciada com 0's.
M_GPU[2][(c1 + 1) · (c2 + 1)] ← {0};
// Matriz de n × (c1 + 1) × (c2 + 1) bits iniciada com 0's.
U_GPU[n][c1 + 1][c2 + 1] ← {0};
para i ← 1 até n faça
    kernelParallelKP2Solver⟨ qtdeB, qtdeT ⟩(i, c1, c2, p, w, M_GPU, U_GPU);
z ← M_GPU[n & 1][c1 · (c2 + 1) + c2];
U ← U_GPU;        // Copia U da memória da GPU para a da CPU.
Opt[n] ← {0};              // Vetor de n bits iniciado com 0's.
para i ← n até 1 faça
    se (U[i][c1][c2] = 1) então
        c1 ← c1 − w_{i,1};
        c2 ← c2 − w_{i,2};
        Opt[i] ← 1;
retorna z, Opt.
```

Algoritmo 3: kernelParallelKP2Solver ($i, c_1, c_2, \boldsymbol{p}, \boldsymbol{w}, M_{GPU}, U_{GPU}$)

Entrada: Índice i do objeto; capacidades c_1 e c_2 das dimensões da mochila; vetor $\boldsymbol{p}$ de lucros dos objetos; matriz $\boldsymbol{w}$ de pesos dos objetos; matriz M_{GPU} de estados; matriz U_{GPU} de *bits*.

```
tId ← obtemThreadId() + obtemBlocoId() · obtemQtdeThreadsBloco();
passo ← obtemQtdeBlocos() · obtemQtdeThreadsBloco();
para j ← tId ; j < (c1 + 1) · (c2 + 1) ; j ← j + passo faça
    c'_1 ← ⌊j/(c2 + 1)⌋;
    c'_2 ← j % (c2 + 1);
    M_GPU[i & 1][j] ← M_GPU[(i − 1) & 1][j];
    se (c'_1 ⩾ w_{i,1}) e (c'_2 ⩾ w_{i,2}) então
        c''_1 ← c'_1 − w_{i,1};
        c''_2 ← c'_2 − w_{i,2};
        se (M_GPU[i & 1][j] ⩽ M_GPU[(i − 1) & 1][c''_1 · (c2 + 1) + c''_2] + p_i) então
            M_GPU[i & 1][j] ← M_GPU[(i − 1) & 1][c''_1 · (c2 + 1) + c''_2] + p_i;
            U_GPU[i][c'_1][c'_2] ← 1;
```

3.3. Resolução em paralelo de várias instâncias *KP1* e *KP2*

Algumas abordagens para resolução de instâncias do *MKP*, com $m \gg 2$ dimensões, podem exigir a resolução de várias instâncias de menor porte do *MKP*, tipicamente com $1 \leqslant m \leqslant 3$. Uma possibilidade, em desenvolvimento, é um algoritmo baseado nas técnicas de geração de colunas associado à enumeração implícita de soluções (algoritmo *Branch-and-Price*), no qual o subproblema de *pricing* (geração de colunas) reduz-se à resolução de várias instâncias do *KP1* ou do *KP2*. Assim, é proposto a seguir um algoritmo para a resolução eficiente de várias instâncias do *KP2* ao mesmo tempo.

Seja (*KP2*)^k o problema de se resolver, de uma só vez, $k > 1$ instâncias do *KP2* com igual quantidade n de objetos. Nessa extensão, c_1^k e c_2^k representam as capacidades da mochila na k-ésima instância e $\boldsymbol{p^k}$ e $\boldsymbol{w_1^k}$ são as matrizes de lucros e pesos dos objetos, respectivamente. A resolução de (*KP2*)^k é equivalente à resolução de k instâncias do *KP2*. Uma opção de resolução paralela para o (*KP2*)^k seria apenas executar o Algoritmo 2 com cada uma das k instâncias do *KP2*, uma de cada vez. Contudo, é possível ajustar o Algoritmo 2 ao (*KP2*)^k, de forma a gerar um maior nível de paralelismo. Essa nova estratégia consiste em executar o *kernel* uma única vez para todas as k mochilas. Assim, dado um índice i de um objeto, computa-se a matriz de estados para as k mochilas com apenas uma chamada ao *kernel*. Dessa forma, consegue-se aumentar a carga de trabalho na *GPU* e diminuir as operações sequenciais na *CPU*, gerando-se um menor tempo de processamento total para a resolução do (*KP2*)^k.

Algoritmo 4: multiParallelKP2Solver $(n, k, \boldsymbol{c_1}, \boldsymbol{c_2}, \boldsymbol{p}, \boldsymbol{w}, qtdeB, qtdeT)$

Entrada: Quantidade n de objetos; quantidade k de mochilas; vetores $\boldsymbol{c_1}$ e $\boldsymbol{c_2}$ de capacidades das mochilas; matrizes $\boldsymbol{p}$ e $\boldsymbol{w}$ de lucros e pesos dos objetos; quantidades $qtdeB$ e $qtdeT$ de blocos e *threads* lançadas em cada bloco na *GPU*.

Saída: Vetor de valores máximos Z. Matriz Opt de *bits* que indica os objetos selecionados para cada mochila.

1 $S[k+1] \leftarrow \{0\}$; `// Vetor de inteiros (k+1) posições iniciado com 0's`
2 **para** $i \leftarrow 1$ **até** k **faça**
3 $\quad S[i] \leftarrow S[i-1] + (c_1^i + 1) \cdot (c_2^i + 1)$;
4 $M_{GPU}[2][S[k]] \leftarrow \{0\}$; `// 2×S[k] estados iniciados com 0's.`
5 $U_{GPU}[n][S[k]] \leftarrow \{0\}$; `// n×S[k] bits iniciados com 0's.`
6 $S_{GPU} \leftarrow S$; `// Copia de S para a memória da GPU.`
7 **para** $i \leftarrow 1$ **até** n **faça**
8 $\quad$ kernelMultiParallelKP2Solver⟨qtdeB, qtdeT⟩$(k, i, \boldsymbol{c_2}, \boldsymbol{p}, \boldsymbol{w}, M_{GPU}, U_{GPU}, S_{GPU})$;
9 $Z_{GPU}[k] \leftarrow \{0\}$; `// Vetor de k posições iniciado com 0's.`
10 $Opt_{GPU}[k][n] \leftarrow \{0\}$; `// Matriz de k × n bits iniciada com 0's.`
11 $maxTporB \leftarrow obtemMaximoThreadsPorBloco()$;
12 kernelBacktrackingKP2⟨ $\left\lceil \frac{k}{maxTporB} \right\rceil, \min(k, maxTporB)$ ⟩$(n, k, \boldsymbol{c_1}, \boldsymbol{c_2}, \boldsymbol{w}, M_{GPU}, U_{GPU}, S_{GPU}, Z_{GPU}, Opt_{GPU})$;
13 $Z \leftarrow Z_{GPU}$;
14 $Opt \leftarrow Opt_{GPU}$;
15 **retorna** Z, Opt.

Os Algoritmos 4 a 6 detalham essa abordagem. Esses algoritmos também podem ser facilmente adaptados para a resolução do (*KP1*)^k, o problema de se resolver, de uma só vez, $k > 1$ instâncias da versão unidimensional *KP1*, também com a restrição de que todas tenham igual quantidade n de objetos.

Para a resolução do (*KP2*)^k foi utilizado um novo vetor S, onde $S[i]$ contém a soma das dimensões das capacidades das mochilas até i, isto é, $S[i] = \sum_{\ell=1}^{i}(c_1^\ell + 1) \cdot (c_2^\ell + 1)$, para $0 \leqslant i \leqslant k$ (passos 1–3 do Algoritmo 4). Esse vetor especifica as posições iniciais e finais da concatenação dos estados de cada uma das mochilas, isto é, $S[k-1]$ e $S[k] - 1$ são, respectivamente, as posições iniciais e finais na nova matriz de estados para k-ésima mochila. As matrizes M_{GPU} e U_{GPU} têm sua segunda dimensão alterada para $S[k]$, que é o tamanho da concatenação das linhas da matriz de estados de todas as k mochilas (passos 4 e 5). O passo 6 apenas especifica que o vetor S é copiado para a memória da *GPU*. Os passos 7 e 8 realizam a chamada ao *kernel*, o qual computa uma linha inteira da matriz de estados. Nos passos 9 e 10 são definidos o vetor Z_{GPU} (os valores máximos da solução de cada uma das k instâncias do *KP2*) e a matriz de *bits* Opt (subconjunto de objetos nas soluções ótimas).

Uma particularidade dessa nova abordagem é a fase de *backtracking*, realizada de forma paralela (passo 12). Como a resolução de cada mochila implica em modificações em elementos diferentes nas matrizes Z e Opt, pode-se lançar uma *thread* para cada uma das k instâncias. Em cada uma delas, o valor máximo da solução é resgatado e armazenado em Z e os objetos recuperados são indicados pelos *bits* na matriz Opt.

Algoritmo 5: kernelMultiParallelKP2Solver $(k, i, \boldsymbol{c_2}, \boldsymbol{p}, \boldsymbol{w}, M_{GPU}, U_{GPU}, S_{GPU})$

Entrada: Quantidade de mochilas k; objeto i a ser processado; vetor $\boldsymbol{c_2}$ de capacidades; matriz $\boldsymbol{p}$ de lucros; matriz $\boldsymbol{w}$ de pesos; matriz M de estados; matriz U de *bits*; vetor S de posições iniciais.

```
1  tId ← obtemThreadId() + obtemBlocoId() · obtemQtdeThreadsBloco();
2  passo ← obtemQtdeBlocos() · obtemQtdeThreadsBloco();
3  id ← 1;
4  para j ← tId ; j < S_GPU[k] ; j ← j + passo faça
5      enquanto (j ⩾ S_GPU[id]) faça id ← id + 1;
6      c'_1 ← ⌊(j − S_GPU[id − 1])/(c_2^id + 1)⌋;
7      c'_2 ← (j − S_GPU[id − 1])%(c_2^id + 1);
8      M_GPU[i & 1][j] ← M_GPU[(i − 1) & 1][j];
9      se ((c'_1 ⩾ w_{i,1}^id) e (c'_2 ⩾ w_{i,2}^id)) então
10         pos ← S_GPU[id − 1] + (c'_1 − w_{i,1}^id) · (c_2^id + 1) + (c'_2 − w_{i,2}^id);
11         se (M_GPU[i & 1][j] < M_GPU[(i − 1) & 1][pos] + p_i^id) então
12             M_GPU[i & 1][j] ← M_GPU[(i − 1) & 1][pos] + p_i^id;
13             U_GPU[i][j] ← 1;
```

O *kernel* para o processamento de uma linha da matriz de estados, dado um objeto i, é descrito no Algoritmo 5. O seu funcionamento é similar ao das outras versões de *kernel* descritas anteriormente neste artigo. Nesse algoritmo foi adicionada a variável id, que especifica a qual mochila pertence o estado que está sendo processado. Inicialmente id é inicializada com 1 e, conforme a variável de controle j (passo 4) é incrementada, o novo estado j pode não mais se referir à mochila id, mas sim a alguma mochila posterior

à id. Assim, o passo 5 realiza o trabalho de manter sempre o id correto com relação ao estado j que está sendo computado. Os demais passos realizam basicamente o mesmo procedimento que o Algoritmo 3. A diferença básica está no uso da expressão $(j - S[id-1])$ para o cálculo dos valores das capacidades c'_1 e c'_2 (passos 6 e 7). Essa expressão equivale ao produto $(c_1^{id}+1)\cdot(c_2^{id}+1)$, visto que $S[id-1]$ especifica a posição inicial na matriz de estados referente à id-ésima instância do *KP2*.

O *kernel* que realiza o procedimento de *backtracking* da programação dinâmica é detalhado no Algoritmo 6. O passo 1 define a variável id que armazena o índice da instância *KP2* relativa à *thread*. Os passos 2 e 3 encerram o processamento caso sejam lançadas mais *threads* que a quantidade de instâncias. O passo 4 recupera a capacidade da mochila id por meio do vetor S. Os passos 6 à 10 recuperam os objetos da mesma forma que os passos 8 à 12 do Algoritmo 2.

Algoritmo 6: kernelBacktrackingKP2 $(n, k, \boldsymbol{c_1}, \boldsymbol{c_2}, \boldsymbol{w}, M_{GPU}, U_{GPU}, S_{GPU}, Z_{GPU}, Opt_{GPU})$

Entrada: Quantidade n de objetos; quantidade k de mochilas; vetores $\boldsymbol{c_1}$ e $\boldsymbol{c_2}$ de capacidades; matriz $\boldsymbol{w}$ de pesos dos objetos; matriz M_{GPU} de estados; matriz U_{GPU} de *bits*; vetor S_{GPU} de posições iniciais; vetor Z_{GPU} de valores máximos; matriz Opt_{GPU} de *bits*.

1 $id \leftarrow 1 + obtemThreadId() + obtemBlocoId() \cdot obtemNumThreadsPorBloco()$;
2 **se** $(id > k)$ **então**
3 **retorna**;
4 $Z_{GPU}[id] \leftarrow M_{GPU}[n \,\&\, 1][S_{GPU}[id-1] + c_1^{id} \cdot (c_2^{id}+1) + c_2^{id}]$;
5 $c_{aux} \leftarrow c_2^{id} + 1$;
6 **para** $i \leftarrow n$ **até** 1 **faça**
7 **se** $(U_{GPU}[i][S[id-1] + c_1^{id} \cdot c_{aux} + c_2^{id}] = 1)$ **então**
8 $c_1^{id} \leftarrow c_1^{id} - w_{i,1}^{id}$;
9 $c_2^{id} \leftarrow c_2^{id} - w_{i,2}^{id}$;
10 $Opt_{GPU}[id][i] \leftarrow 1$;

4. Testes comparativos

Nesta seção são descritos alguns dos testes computacionais realizados com o intuito de mensurar e verificar a eficiência dos algoritmos apresentados neste artigo, com relação ao tempo de processamento. Todos os testes foram realizados em computador com processador "Intel(R) Xeon(R) CPU @ 2.20GHz" e 12*GB* de memória *RAM*. A *GPU* utilizada foi uma *Tesla T4* com cerca de 15843*MB* de memória. A linguagem de programação *C++14* foi usada na implementação dos algoritmos, com o auxílio da *API CUDA* para na codificação dos algoritmos paralelos. Foi utilizado o *GCC 7.5.0* para a compilação do Algoritmo 1 e o *NVCC 9.2* para a compilação dos demais algoritmos. A *flag O3* foi sempre utilizada como parâmetro na compilação.

Nos testes foram utilizadas duas classes de instâncias da base *2DPackLib* [Iori et al. 2022]. A classe *A*, proposta por [Macedo et al. 2010], contém instâncias do *2D-CSP – Two-dimensional Cutting Stock Problem*. A classe *MSB*, proposta por [Mesyagutov et al. 2012], contém instâncias do *2D-OPP – Two-dimensional Orthogonal Packing Feasibility Problem*. As instâncias de ambas as classes foram adaptadas para o *KP2* como descrito a seguir: as capacidades c_1 e c_2 são iguais, respectivamente, aos va-

lores H e W (altura e largura da placa original); os pesos $w_{i,1}$ e $w_{i,2}$ e o valor p_i do objeto i são iguais, respectivamente, à altura, à largura e à área do item da instância original. As soluções ótimas para as instâncias do *KP2* assim obtidas, não necessariamente correspondem a soluções ótimas para as respectivas instâncias do *2D-OPP* ou do *2D-CSP*. Contudo, o valor das soluções para as instâncias do *KP2* são limites inferiores para os valores ótimos para as respectivas instâncias do *2D-OPP* e do *2D-CSP*. De agora em diante, as referências às classes *A* e *MSB* referem-se às versões adaptadas para o *KP2*.

A Figura 1 apresenta o tempo de processamento do algoritmo sequencial (barra laranja vertical) em contraste com o tempo de processamento da versão paralela (barra azul vertical), em testes com parâmetros $qtdeB$=$qtdeT$=64, com as instâncias A_i, $i = 1, \ldots, 43$, da classe A. As instâncias possuem capacidades $c_1 \in \{1220, 2080, 2100\}$ (linha verde) e $c_2 \in \{2470, 2550, 2750\}$ (linha vermelha). A quantidade de objetos de cada uma é mostrada acima da respectiva barra laranja.

No caso particular da instância A_{33} (n=134, c_1=2550 e c_2=2100), o tempo de processamento do Algoritmo 1 foi de $6287,11$ms e o Algoritmo 2 consumiu $282{,}71$ms, com um *speedup* de $22{,}24$. Esse ganho significativo no tempo de execução também pode ser observado em várias outras instâncias. As instâncias que apresentaram tempos de processamento muito próximos, em ambos os algoritmos, são instâncias de porte relativamente pequeno e, portanto, não necessitam de muito processamento para a sua resolução. Em alguns desses casos, o algoritmo paralelo se tornou até mais lento que o sequencial, devido ao tempo gasto para lançar as *threads* na *GPU* ser próximo, ou mesmo superior, ao tempo de se resolver a instância de forma sequencial na *CPU*.

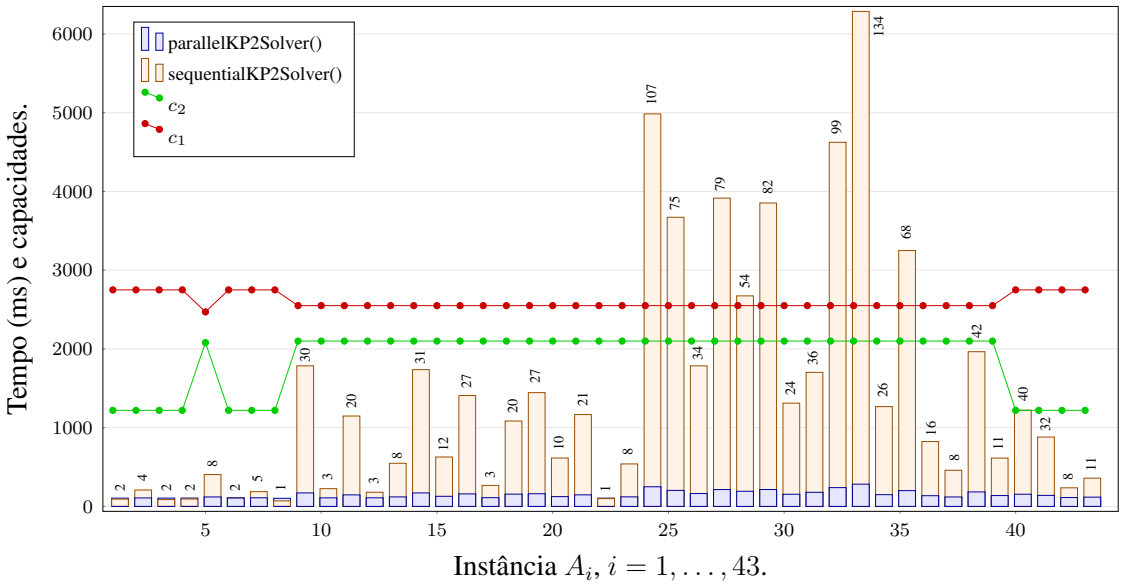

Figura 1. Parâmetros e tempos de processamento com instâncias da classe *A*.

A classe MSB contém 630 instâncias, cada uma com $n = 20$ objetos e capacidades $c_1 = c_2 = 1000$. A Tabela 1 apresenta o tempo total consumido na resolução dessas instâncias. Nas duas primeiras colunas ("Serial") são relativas à execução do Algoritmo 1. As colunas "Total" e "Média" mostram, respectivamente, o tempo total e médio consumidos na resolução das 630 instâncias. A terceira e a quarta colunas ("Recursos") mostram

as quantidades de blocos e de *threads* por bloco usadas na execução dos algoritmos paralelos. A quinta à sétima coluna ("Paralelo") são relativas ao Algoritmo 2. As duas primeiras colunas deste bloco são similares às do bloco "Serial" e a seguinte mostra o *speedup* médio do Algoritmo 2 em relação ao Algoritmo 1. Como essas instancias são de porte relativamente pequeno, o ganho do algoritmo paralelo não é significativo para essa classe. Por fim, as últimas três colunas ("Multi Paralelo") são relativas ao esforço do Algoritmo 4 para resolver todas as 630 instancias ao mesmo tempo. A última coluna mostra o *speedup* médio do Algoritmo 4, também em relação ao Algoritmo 1.

Tabela 1. 630 Instâncias MSB – 20 objetos e capacidades iguais a 1000.

Serial		**Recursos**		**Paralelo**			**Multi Paralelo**		
Total	**Média**	**Blocos**	***Threads***	**Total**	**Média**	***Speedup***	**Total**	**Média**	***Speedup***
168889,91	268,08	32	32	139682,39	221,71	1,21	10978,94	17,43	15,38
		32	64	133420,01	211,78	1,26	5870,54	9,32	28,76
		64	64	127905,18	203,02	1,32	3358,43	5,33	50,30
		64	128	124914,43	198,28	1,35	2027,44	3,22	83,25
		128	128	125278,91	198,85	1,35	1393,80	2,21	121,30
		128	256	123185,03	195,53	1,37	1100,34	1,75	153,18
		256	256	123837,77	196,57	1,36	1098,56	1,74	154,07

Os tempos de resolução das instâncias, expressos na Figura 1 e na Tabela 1, incluem tanto o tempo de processamento quanto o tempo de alocação e transferência de memória. Além disso, para cada uma das instâncias, o tempo de resolução considerado foi a média aritmética entre 10 execuções. É evidente que a decisão de resolver todas as instâncias de uma só vez permitiu um maior nível de paralelismo, aumentando o trabalho que cada *thread* realiza e diminuindo o número de chamadas ao *kernel*, feitas pela *CPU*, quando comparado à estratégia de resolver uma instância de cada vez. Esse último fato foi o responsável pelo grande ganho de eficiência sobre os Algoritmos 2 e 4.

5. Conclusões

Na Seção 3 foram apresentados algoritmos para a resolução sequencial e paralela do *KP2* (Algoritmos 1 e 2). Além disso, também foi proposto o Algoritmo 4, destinado à resolução da extensão $(KP2)^k$ apresentada na Seção 3.3. Esse algoritmo possuí como princípio a realização da maior quantidade possível de trabalho na *GPU*. Ressalte-se ainda que as estratégias apresentadas neste trabalho podem ser facilmente adaptadas para resolver também o *MKP* com $m > 2$ dimensões.

A Seção 4 apresenta testes computacionais que mensuraram os tempos de processamento utilizados por cada algoritmo e os comparam segundo as mesmas instâncias. Observou-se que o Algoritmo 2 gerou um ganho de eficiência considerável, na resolução de instâncias de maior porte, quando comparado ao Algoritmo 1. Os resultados mostram que, dada a forma como o Algoritmo 2 foi construído, quanto maior forem as capacidades das dimensões da mochila e menor a quantidade de objetos, maior será esse ganho. Além disso, o Algoritmo 4 foi construído de forma a resolver várias instâncias do *KP2* de uma só vez, aumentando o trabalho de cada *thread* lançada na *GPU*. Assim, conseguindo reduzir o tempo total de processamento de todas as instâncias.

Durante a execução deste trabalho, identificou-se uma proposta de estudo futuro. Essa proposta envolve o desenvolvimento de algoritmos para a resolução de problemas

da mochila, por meio da técnica de programação dinâmica, com suporte ao processamento paralelo permitido pelas *GPU*, considerando uma abordagem *top–down*, diferente da apresentada neste artigo que é a *bottom–up*. A abordagem *top–down* permite que apenas os subproblemas realmente utilizados, de forma direta ou indireta, para a resolução do problema final, sejam computados. Embora esse fato possa indicar um ganho de eficiência, de tempo de execução e de memória, do método *top–down* quando comparado ao *bottom–up*, a sua implementação pode ser complexa. Essa dificuldade deve-se à característica recursiva da abordagem *top–down*. Além disso, a fase de *backtracking* também poderia se tornar complicada caso não fosse armazenado um *bit* (*flag*) para cada subproblema, como é feito nos algoritmos deste artigo.

Referências

Bansal, J. C. and Deep, K. (2012). A Modified Binary Particle Swarm Optimization for Knapsack Problems. *Appl. Math. Comput.*, 218:11042–11061.

Bellman, R. (1957). *Dynamic Programming*. Princeton University Press, Princeton, NJ, USA, 1 edition.

Garland, M., Le Grand, S., Nickolls, J., Anderson, J., Hardwick, J., Morton, S., Phillips, E., Zhang, Y., and Volkov, V. (2008). Parallel Computing Experiences with CUDA. *IEEE Micro*, 28(4):13–27.

Gavish, B. and Pirkul, H. (1982). Allocation of Databases and Processors in a Distributed Computing System. In Akoka, J., editor, *Management of Distributed Data Processing*, volume 31, pages 215–231. North-Holland.

Gilmore, P. C. and Gomory, R. E. (1966). The Theory and Computation of Knapsack Functions. *Operations Research*, 14(6):1045–1074.

Iori, M., de Lima, V. L., Martello, S., and Monaci, M. (2022). 2dpacklib: a two-dimensional cutting and packing library. *Optimization Letters*, 16:471–480.

Kellerer, H., Pferschy, U., and Pisinger, D. (2004). *Knapsack Problems*. Springer.

Lorie, J. H. and Savage, L. J. (1955). Three problems in capital rationing. *Journal of Business*, 28(4):229–239.

Macedo, R., Alves, C., and Carvalho, J. (2010). Arc-flow model for the two-dimensional guillotine cutting stock problem. *Computers & OR*, 37:991–1001.

Mesyagutov, M., Scheithauer, G., and Belov, G. (2012). Lp bounds in various constraint programming approaches for orthogonal packing. *Computers & Operations Research*, 39(10):2425–2438.

Nickolls, J., Buck, I., Garland, M., and Skadron, K. (2008). Scalable Parallel Programming with CUDA: Is CUDA the parallel programming model that application developers have been waiting for? *Queue*, 6(2):40–53.

Pfeffer, A. (2007). Sampling with memoization. In *Proc. of the 22nd National Conference on Artificial Intelligence - Volume 2*, AAAI'07, pages 1263–1270. AAAI Press.

Shih, W. (1979). A Branch and Bound Method for the Multiconstraint Zero-One Knapsack Problem. *Journal of the Operational Research Society*, 30(4):369–378.