MEC-SETEC

INSTITUTO FEDERAL MINAS GERAIS - *Campus Formiga*

Curso de Ciência da Computação

Uso da plataforma CUDA na metaheurística GRASP para resolução do problema da mochila.

Ronan Nunes Campos

Orientador: Prof. Me. Walace de Almeida Rodrigues

Formiga - MG

2018

RONAN NUNES CAMPOS

Uso da plataforma CUDA na metaheurística GRASP para resolução do problema da mochila.

Trabalho de Conclusão de Curso apresentado ao Instituto Federal Minas Gerais - Campus Formiga, como requisito parcial para a obtenção do título de Bacharel em Ciência da Computação.

Orientador: Prof. Me. Walace de Almeida Rodrigues.

Formiga - MG

2018

Campos, Ronan Nunes.

004 Uso da plataforma CUDA na metaheurística GRASP para resolução do problema da mochila / Ronan Nunes Campos. -- Formiga : IFMG, 2018.

55p.: il.

Orientador: Prof. Msc Walace de Almeida Rodrigues

Trabalho de Conclusão de Curso – Instituto Federal de Educação, Ciência e Tecnologia de Minas Gerais – *Campus* Formiga.

1. GRASP. 2. CUDA. 3. Problema da mochila. 4. Metaheurística. I. Título.

CDD 004

Ficha catalográfica elaborada pela Bibliotecária Me..Naliana Dias Leandro CRB6-1347

RONAN NUNES CAMPOS

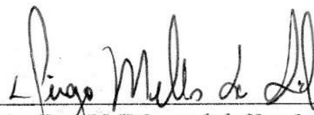
USO DA PLATAFORMA CUDA NA METAHEURÍSTICA GRASP PARA RESOLUÇÃO DO PROBLEMA DA MOCHILA

Trabalho de Conclusão de Curso apresentado ao Instituto Federal de Minas Gerais-Campus Formiga, como Requisito parcial para obtenção do título de Bacharel em Ciência da Computação.

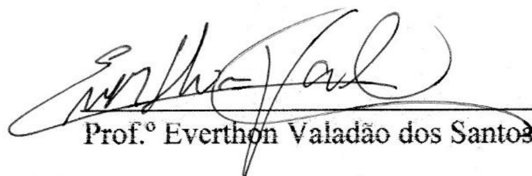
Aprovado em: 05 de Junho de 2018.

BANCA EXAMINADORA

Prof.º Walace de Almeida Rodrigues



Prof.º Diego Mello da Silva



Prof.º Everthon Valadão dos Santos

Resumo

A computação paralela é uma alternativa interessante para agilizar o processamento de tarefas que envolvem muitos cálculos. Atualmente, com a popularização e o avanço das placas de vídeo, é cada vez mais comum encontrar computadores domésticos que trazem pelo menos uma GPU (Graphic Processing Unit) no seu *hardware*. A CUDA *(Compute Unified Device Architecture*) é a plataforma de computação paralela desenvolvida pela NVIDIA, a qual vem ganhando notoriedade por seu desempenho. O presente trabalho tem como objetivo avaliar o desempenho da arquitetura CUDA na resolução do *Knapsack Problem 01* ou Problema da Mochila 01 (PM), um problema clássico de otimização combinatória, que pode ser enunciado como a necessidade de carregar uma mochila com capacidade limitada, com um conjunto de objetos com peso e benefício diferentes, maximizando o benefício total da mochila respeitando seu limite de carga. A abordagem adotada para avaliar o desempenho da arquitetura CUDA foi implementar uma metaheurística GRASP para resolver o PM e comparar os resultados, considerando três experimentos distintos: no primeiro, o GRASP é executado sequencialmente na CPU; no segundo, o GRASP é executado de forma paralela na CPU, utilizando *threads*; no terceiro, o GRASP é executado paralelamente na GPU. A avaliação compara estatisticamente os resultados obtidos quanto ao tempo de execução nos três experimentos.

Palavras-chave: GRASP, CUDA, problema da mochila, metaheurística.

Abstract

Parallel computing is an interesting alternative to accelerate the processing of tasks involving many calculations. Today, with the popularization and advance of video cards, it is common to find home computers that bring at least a GPU (Graphic Processing Unit) on your hardware. CUDA (Compute Unified Device Architecture) is the parallel computing platform developed by NVIDIA, which has been gaining notoriety for its performance. The present work aims to evaluate the performance of the CUDA architecture in the resolution of the Knapsack Problem 01 (KP), a classic problem of combinatorial optimization, which can be stated as how the best way to load a knapsack with limited capacity, using a set of objects of different weight and benefit, maximizing the total benefit of the knapsack respecting its capacity. The approach adopted to evaluate the performance of the CUDA architecture was to implement a GRASP metaheuristic to solve the PM and compare the results considering three different experiments: in the first, GRASP is executed sequentially in the CPU; in the second, GRASP runs concurrently on the CPU, using threads; in the third, GRASP runs in parallel on the GPU. The evaluation compares statistically the results obtained for the runtime in the three experiments.

Key words: GRASP, CUDA, Knapsack problem, metaheuristics.

Lista de Figuras

Figura 1 - Pseudocódigo GRASP 22
Figura 2 - Pseudocódigo fase de construção GRASP 22
Figura 3 - Fase de busca GRASP 23
Figura 4 - Construção da RCL 24
Figura 5 - Arquitetura CPU vs GPU (ALVES, 2014) 26
Figura 6 - Arquitetura típica de uma GPU CUDA (DAVID, WEN-MEI, 2011) 27
Figura 7 - Camadas da API CUDA 28
Figura 8 - Escalabilidade de um programa Multithread CUDA 30
Figura 9 - Organização da grade CUDA (DAVID, WEN-MEI, 2011) 31
Figura 10 - Distribuição de Memória GPU 33
Figura 11 - Chamada do Kernel e Assinatura da função 36
Figura 12 - Propriedades da GPU 37
Figura 13 - Passos Básicos do GRASP 39
Figura 14 - Chamada da função que cria a solução inicial 39
Figura 15 - Construção da solução inicial 40
Figura 16 - Busca Local GRASP 41
Figura 17 - Método Roleta 41
Figura 18 - Criação das Pthreads 42
Figura 19 - Iniciando cada pthread para rodar o GRASP na CPU 42
Figura 20 - Helper CUDA 43

Lista de Tabelas

Tabela 1 - Instâncias de teste. 37
Tabela 2 - Exemplo da estrutura de dados ‘item’ preenchida com 5 posições. 38
Tabela 3 - Parâmetros GRASP Sequencial 44
Tabela 4 - Parâmetros GRASP Paralelo CPU 44
Tabela 5 - Parâmetros GRASP Paralelo GPU 45
Tabela 6 - Resultados GRASP Sequencial 46
Tabela 7 - Resultados GRASP Paralelo CPU 47
Tabela 8 - Resultados GRASP Paralelo GPU 100 Itens 49
Tabela 9 - Resultados GRASP Paralelo GPU 500 Itens 49
Tabela 10 - Resultados GRASP Paralelo GPU 1000 Itens 49
Tabela 11 - Resultados GRASP Paralelo GPU 2000 Itens 50

Lista de Gráficos

Gráfico 1 - Tempo x Itens GRASP Sequencial 47
Gráfico 2 - Tempo x Itens GRASP Paralelo CPU 48
Gráfico 3 - Tempo x Itens GRASP Paralelo GPU 50
Gráfico 4 - Comparação dos Testes 51

Sumário

1. **Introdução** ... 17
1.1. **Objetivo** ... 18
1.1.1. **Objetivo Geral** ... 18
1.1.2. **Objetivos Específicos** ... 18
1.2. **Estrutura do Trabalho** ... 19
2. **Fundamentação Teórica** ... 19
2.1. **O Problema da Mochila** ... 19
2.2. **Metaheurística e Heurística** ... 20
2.3. **GRASP** ... 21
2.3.1. **Fase de Construção** ... 22
2.3.2. **Fase de Busca Local** ... 23
2.3.3. **Construção da RCL** ... 24
2.3.4. **GRASP Paralelo** ... 25
2.4. **GPU – Graphics Processing Unit** ... 25
2.5. **CUDA** ... 27
2.5.1. **API CUDA** ... 28
2.5.2. **Modelo de Programação** ... 29
2.5.3. Kernels ... 30
2.5.4. **Hierarquia de Thread** ... 31
2.5.5. **Hierarquia de Memória** ... 32
2.6. **CUDA em C** ... 33
2.6.1. **Compilação com NVCC** ... 34
2.6.2. **Fluxo de Execução** ... 34
2.6.3. **Programando com CUDA em C** ... 35
3. **Metodologia** ... 36
3.1. **Materiais** ... 36
3.2. **Métodos** ... 38
4. **Experimentos Computacionais** ... 44
4.1. **Resultados GRASP Sequencial** ... 46
4.2. **Resultados GRASP paralelizado CPU** ... 47
4.3. **Resultados GRASP Paralelo GPU** ... 48
4.4. **Limitações de Hardware** ... 51
5. **Conclusão** ... 52
Referências ... 54

1. Introdução

O Problema da Mochila na sua variante 01 é um problema clássico de otimização combinatória que possui uma ampla gama de aplicações, tais como dimensionamento de cargas, orçamento de capital, alocação de recursos em sistemas distribuídos, entre outros. Na sua versão binária, o Problema da Mochila pode ser definido como uma situação em que é necessário preencher uma mochila com objetos de diferentes pesos e valores. O objetivo é que se preencha a mochila com o maior valor possível, não ultrapassando o peso máximo.

Apesar da simplicidade de seu enunciado, esse problema é um dos 21 problemas NP-Difícil listados por Richard Karp, exposto em 1972 (CARVALHO; SILVA, 2016). Para instâncias reais de grande porte, quando o conjunto de objetos ou restrições do problema cresce, algoritmos que buscam soluções exatas são computacionalmente inviáveis. Para a obtenção de soluções viáveis que sejam de boa qualidade, no sentido de aproximarem-se do ótimo, faz-se necessária a utilização de algoritmos de aproximação ou heurísticos.

As heurísticas são técnicas que, em geral, são capazes de obter rapidamente boas soluções para diversos problemas. Entretanto, diferentemente dos algoritmos de aproximação, não há garantia alguma sobre a qualidade de tais soluções (DANTAS, 2016). Neste sentido muitas heurísticas para otimização combinatória seguem as diretrizes de metaheurísticas, tais como: Algoritmos Genéticos, Simulated Annealing, Busca Tabu e Procedimento de Busca Guloso, Aleatório e Adaptativo (GRASP) (RESENDE; SILVA, 2013).

Apesar de serem abordagens viáveis para resolver problemas difíceis, mesmo heurísticas ou metaheurísticas podem ser bastante custosas, motivando a utilização de técnicas para redução do tempo de execução ou para melhorar a qualidade das soluções como, por exemplo, estratégias de paralelização (DANTAS, 2016).

A metaheurística GRASP *Greedy Randomized Adaptive Search Procedure* é um método multipartida para otimização combinatória que aplica a busca local repetidamente a partir de soluções construídas por algoritmos gulosos aleatórios. A ideia foi originalmente desenvolvida por Feo e Resende (RAIDL, 2006) e deu origem a uma metaheurística bastante adequada para implementação em processadores paralelos procurando conciliar a simplicidade conceitual das heurísticas gulosas com a limitação destas gerarem a mesma e única solução cada vez que acionadas.

Diferentes formas de arquiteturas paralelas têm sido estudadas através dos anos, sempre buscando o aumento da eficiência e a diminuição do custo. Dentre as opções que existem atualmente tem-se desde supercomputadores a *clusters*. Impulsionados pela necessidade crescente e o surgimento de um novo ramo da ciência focado na busca de poder de processamento, alguns pesquisadores começaram a buscar novas opções de dispositivos para serem utilizados nas novas arquiteturas paralelas (P. ROCHA; F FILHO, 2010)

Uma dessas novas opções de dispositivos é a Unidade de Processamento Gráfico ou *Graphics Processing Unit* - GPU. Primeiramente criada apenas para processamentos gráficos em tarefas de renderização de imagens e cálculos vetoriais, entre outros, a GPU adquiriu um grande poder de processamento e a possibilidade de ser programável (P. ROCHA; F FILHIO, 2010). Menos de um ano após a NVIDIA ter cunhado o termo GPU, em 1999 lançando sua primeira GPU, a GeForce 256 (NVIDIA, 2018), artistas e desenvolvedores de jogos não eram os únicos a realizar trabalhos inovadores com esta tecnologia: pesquisadores estavam tirando proveito de seu excelente desempenho de ponto flutuante. Nascia o movimento da GPU de Propósito Geral (GPGPU) (NVIDIA, 2018).

Como no início apenas APIs Gráficas, por exemplo o OpenGL, poderiam ser utilizadas para a programação na GPU, houve uma grande dificuldade na criação de programas que não fossem gráficos devido à falta de abstração do dispositivo, forçando o programador a ter conhecimento da arquitetura e funcionamento da unidade de processamento. Este fato mudou quando em 2006 foi lançada uma nova arquitetura de abstração, CUDA (*Compute Unified Device Architecture*) pela NVIDIA. Esta nova arquitetura possibilitou o uso em conjunto da CPU (Unidade Central de Processamento) e GPU para o processamento de aplicações de propósito geral, sem que o programador necessitasse do entendimento do processamento gráfico (P. ROCHA; F FILHO, 2010).

1.1. Objetivo

Esta seção apresenta os objetivos gerais e específicos do presente trabalho.

1.1.1. Objetivo Geral

Avaliar o uso da plataforma CUDA desenvolvendo três algoritmos (sequencial, paralelo em CPU e paralelo em GPU) utilizando a metaheurística GRASP, para a resolução do problema clássico de otimização combinatória conhecido como Problema da Mochila em sua variante 01 com os mesmos conjuntos de testes realizando experimentos e comparando seus resultados.

1.1.2. Objetivos Específicos

1. Estudo da arquitetura da GPU GT 740m da NVIDIA.
2. Estudo das técnicas de programação paralela em GPU.
3. Estudo da plataforma CUDA.
4. Estudo da metaheurística GRASP.

5. Implementação da metaheurística GRASP paralelo em CPU usando *threads*.

6. Implementação da metaheurística GRASP paralelo em CUDA.

7. Implementação da metaheurística GRASP sequencial.

8. Realizar testes e estudos de casos para comparar estatisticamente a eficiência dos algoritmos metaheurístico GRASP paralelo e sequencial.

9. Descrever e avaliar qualitativamente o trabalho de programação em CUDA.

1.2. Estrutura do Trabalho

O trabalho está organizado como se segue. Capítulo 1 são definidos os objetivos gerais e específicos. Seguindo para o Capítulo 2, apresenta toda a base teórica necessária para o entendimento do trabalho, definindo o Problema da Mochila, Metaheurística e Heurística, GRASP, GPU e a plataforma CUDA. Os materiais e métodos utilizados para o desenvolvimento do trabalho, bem como parte dos códigos são descritos no Capítulo 3. O Capítulo 4 fornece os resultados e análises obtidos com os testes executados. O documento finaliza no Capítulo 5, que contém as considerações finais sobre o Trabalho.

2. Fundamentação Teórica

Nesta seção são apresentados os conceitos necessários para o entendimento do trabalho, mais especificamente o que é o Problema da Mochila, noções gerais sobre Metaheurísticas, com ênfase no GRASP, e funcionamento da arquitetura CUDA e seu uso com linguagem computacional.

2.1. O Problema da Mochila

O Problema da Mochila pode ser enunciado como: Dado a capacidade da mochila $m \geq 0$, a quantidade de itens n e, para cada item i em $\{1,...,n\}$, o valor $v_i \geq 0$ e peso $p_i \geq 0$, encontrar um subconjunto de itens S de $\{1,...,n\}$que maximize o valor $v(S)$ sob a restrição $p(S) \leq m$. Nessa definição, $v(S)$ denota a soma $\sum_{i \in S} vi$ e, analogamente, $p(S)$ denota a soma $\sum_{i \in S} pi$. O objetivo do problema é então encontrar um subconjunto de objetos, o mais valioso possível, que maximize o ganho que é o somatório do valor de cada item, respeitando a capacidade da mochila (CALDAS; ZIVIANI, 2004).

No Problema da Mochila 01 (*01 Knapsack Problem*), cada item pode ser escolhido no máximo uma vez para compor a solução, enquanto que no Problema da Mochila Limitado (*Bounded Knapsack Problem*) temos uma quantidade limitada para cada tipo de

item. O Problema da Mochila com Múltipla Escolha (*Multiple-choice Knapsack Problem*) ocorre quando os itens devem ser escolhidos de classes disjuntas, e se várias mochilas são preenchidas simultaneamente temos o Problema da Mochila Múltipla (*Multiple Knapsack Problem*). A forma mais geral é o Problema da Mochila com múltiplas restrições (*Multi-constrained Knapsack Problem*) o qual é basicamente um problema de programação inteira geral com coeficientes positivos (CALDAS; ZIVIANI, 2004).

O Problema da Mochila possui muitas variantes e pertence à classe dos problemas NP-Difícil. Isso significa que é incerto que seja possível desenvolver algoritmos polinomiais para este problema. Porém, a despeito do tempo de execução para o pior caso de todos os algoritmos conhecidos para esses problemas terem custo exponencial, existem diversos exemplos de grandes instâncias que podem ser resolvidos encontrando uma solução ótima em fração de segundos. Neste trabalho todos os algoritmos apresentados resolvem o Problema da Mochila 01, que consiste em escolher *n* itens, tais que o somatório dos valores é maximizado sem que o somatório dos pesos extrapole a capacidade da mochila (CALDAS; ZIVIANI, 2004). Isto pode ser formulado como o seguinte problema de maximização:

$$Maximizar \sum_{j=1}^{n} vj\ xj$$

$$Sujeito \sum_{j=1}^{n} pj\ xj \leq M$$

$$xj \in \{0,1\},\ j = 1...n$$

onde x_j é uma variável binária igual a 1 se *j* deve ser incluído na mochila e 0 caso contrário (CALDAS; ZIVIANI, 2004).

As instâncias para o problema podem ser denotadas como ($p_1, ..., p_n, v_1, ..., v_n, c$) onde *p1* e *v1* representam o peso e valor respectivamente do item e *c* a capacidade da mochila. O tamanho para tais instâncias então pode ser dado por ($2n + 1$) números. A definição de tamanho da instância deve levar em conta o *valor* e não apenas a presença do parâmetro *c (*capacidade da mochila*),* logo o tamanho da instância seria o par (*2n, c*) o que não é uma definição ideal. O tamanho do parâmetro *c* é dado segundo o número de caracteres do valor que é aproximadamente $\log_2$ c (por exemplo: o tamanho do número 75840 é 5 e não 75840), logo o tamanho de uma instância é dado pelo par (*2n, lg c*) (FEOFILOFF, 2015). Como $c = 2^{\log_2 c}$ o tamanho da instância do problema é $2n + 2^{\log_2 c}$ (FEOFILOFF, 2015). Com isso quando a capacidade da mochila cresce, o número de combinações de itens que podem fazer parte da solução cresce de forma exponencial.

2.2. Metaheurística e Heurística

Heurísticas são procedimentos usados em algoritmos para solucionar problemas com um custo computacional aceitável, que acham soluções para um dado problema sem

garantir a otimalidade ou a qualidade da mesma, buscando sempre valores próximos ao ótimo (ANDRADE, 2013). Porém, algoritmos que usam heurísticas acham frequentemente soluções muito próximas da ótima, além de fazê-lo de forma rápida e fácil (CARVALHO; SILVA, 2016).

Um dos procedimentos comumente usados é a busca local, onde o objetivo é procurar no espaço de busca - espaço onde estão presentes todas as possíveis soluções de um dado problema - uma solução melhor, com propriedades de ser uma solução localmente ótima, vizinha da solução corrente (ANDRADE, 2013).

As metaheurísticas surgiram da combinação de métodos básicos de heurísticas, com o objetivo de resolver diversos problemas de otimização combinatória, ou seja, cujo conjunto de soluções é composto por um conjunto finito de valores discretos (ANDRADE, 2013). Elas podem ser construídas da junção de duas ou mais heurísticas, que permite a elevação do número de procedimentos existentes.

A seguir, é apresentado a metaheurística GRASP utilizada neste trabalho.

2.3. GRASP

A Metaheurística GRASP (*Greedy Random Adaptative Search Procedures*) que significa Procedimento de Busca Gulosa Adaptativa Aleatória, foi desenvolvida por Feo e Resende como uma das técnicas promissoras na busca de soluções para problemas de otimização combinatória (RAIDL, 2006). O GRASP é um método iterativo multipartida onde cada iteração consiste em duas fases: a primeira é a fase de construção da solução inicial fatível de forma gulosa, aleatória e adaptativa, seguida da segunda fase, que consiste na fase de refinamento da solução inicial por uma heurística de melhoramento, tipicamente um procedimento de busca local. A solução que se mostrar a melhor dentre todas as iterações é mantida como resultado final (Resende e Ribeiro, 2003). Em sua formulação básica, GRASP é uma metaheurística que não possui nenhum mecanismo de memória, na qual cada iteração é responsável por construir uma nova solução desde o começo, sem tirar proveito de informações provenientes de iterações anteriores (DANTAS, 2016).

Na figura 1 é apresentado o pseudocódigo do algoritmo GRASP:

Figura 1 - Pseudocódigo GRASP

```
Algoritmo 1 GRASP(MaxIter, Seed)
1: ReadInput()
2: for (k ← 1 to MaxIter) do
3:    Solution ← GreedyRandomizedConstruction(Seed)
4:    Solution ← LocalSearch(Solution)
5:    UpdateSolution(Solution, BestSolution)
6: end for
7: return BestSolution
```

Os parâmetros do método são:

- MaxIter: Número de iterações do GRASP a executar;
- *Seed*: Semente geradora de números aleatórios;
- GreedyRandomizedConstruction(): fase de construção da solução inicial factível;
- LocalSearch(): Fase de busca local para refinamento da solução;
- Updatesoluction(): Guarda a melhor solução de todas iterações.

As próximas subseções apresentam detalhes sobre as fases do GRASP.

2.3.1. Fase de Construção

Na fase de construção uma solução factível é construída de forma iterativa, elemento por elemento, escolhidos aleatoriamente a partir da lista de candidatos restrita (*Restrict Candidate List - RCL*). Ela é adaptativa devido a atualização da lista a cada iteração para refletir as mudanças ocorridas pela seleção de elementos anteriores (ALVARENGA; ROCHA, 2006). Na figura 2 é apresentado o pseudocódigo da fase de construção do algoritmo GRASP, que será comentado adiante.

Figura 2 - Pseudocódigo fase de construção GRASP

```
Algoritmo 2 GreedyRandomizedConstruction(Seed)
1: Solution ← ∅
2: Avaliar o custo incremental dos elementos candidatos
3: while (Solution não for uma solução completa) do
4:    Construir a lista de candidatos restrita (RCL)
5:    Selecionar um elemento s da RCL aleatoriamente
6:    Solution ← Solution ∪ s
7:    Reavaliar os custos incrementais
8: end while
9: return Solution
```

Cada iteração desta fase é dividida em três subfases:

- Construção da lista de candidatos restritos (RCL): Nesta fase, uma lista de itens candidatos que podem ser incorporados a solução sem destruir sua factibilidade é criada. A seleção do elemento ao incorporar a lista é feita por uma função de avaliação gulosa e a partir do cálculo de custo incremental de cada elemento candidato, o que leva a formação da lista com os melhores itens.
- Escolha de um elemento aleatório da RCL: O elemento a ser incorporado na solução parcial é escolhido aleatoriamente da RCL, de maneira uniforme.
- Reavaliar o custo incremental: Uma vez que o elemento é incorporado na RCL, a lista é atualizada e os custos incrementais são reavaliados. O custo incremental é o somatório do peso de todos itens que estão presentes na solução.

2.3.2. Fase de Busca Local

As soluções iniciais geradas pela fase de construção não são necessariamente ótimos locais. O que esta fase faz é geralmente melhorar a solução construída, a partir de um procedimento de busca local em vizinhança. Métodos de busca local em problemas de otimização constituem uma família de técnicas baseadas na noção de vizinhança, ou seja, são métodos que percorrem o espaço de busca passando, iterativamente, de uma solução para outra que seja sua vizinha (ALVARENGA; ROCHA, 2006).

A fase de busca local de GRASP aproveita a solução inicial da fase de construção e explora a vizinhança ao redor desta solução. Se uma solução melhor é encontrada, a solução corrente é atualizada, e novamente a vizinhança ao redor da nova solução é pesquisada. O processo se repete até nenhuma solução melhor ser encontrada na vizinhança pesquisada. É preciso ter cuidado em: escolher uma vizinhança apropriada; usar estruturas de dados eficientes para acelerar a busca local; e ter uma boa solução inicial (ALVARENGA; ROCHA, 2006). A figura 3 ilustra, em pseudocódigo, a fase de busca local do GRASP:

Figura 3 - Fase de busca GRASP

Algoritmo 3 LocalSearch(Solution)

```
1: while ( Solution não for localmente ótima) do
2:     Encontrar s' ∈ N(Solution) com f(s') < f(Solution)
3:     Solution ← s'
4: end while
5: return Solution
```

2.3.3. Construção da RCL

Cada iteração da construção da solução inicial, produz uma solução amostrada a partir de uma distribuição desconhecida, cuja média e variância são funções da natureza restritiva da RCL. Destacam-se duas situações resultantes dessa amostragem:

- RCL com único elemento: variância nula e média igual ao valor da solução gulosa;
- RCL com mais elementos: maior variância, média dilui-se de acordo com os valores associados aos elementos, embora melhor solução encontrada supera a média e frequentemente é ótima.

A figura 4 apresenta a construção da RCL. Primeiramente inicializa-se a solução inicial como uma lista de itens vazia, e uma lista de elementos candidatos recebe todo os itens que podem incorporar a solução inicialmente. Então, o custo incremental de cada item presente na lista de candidatos é calculado. Enquanto houver itens candidatos a pertencer a solução, cria-se uma lista de candidatos restritos (RCL) com o item de menor custo incremental (c^{min}) quando $\alpha = 0$, ou com todos itens com custo incremental menor ou igual $c^{min} + \alpha\,(c^{min} + c^{max})$ com $\alpha = 1$. Um elemento pertencente à RCL é sorteado para incorporar a solução. Seguindo, a lista de candidatos é atualizada e o custo incremental de cada item é recalculado.

Figura 4 - Construção da RCL

Algoritmo 4 GreedyRandomizedConstruction(α, *Seed*)

```
1:  Solution ← ∅
2:  Inicializa o conjunto candidato: C ← E
3:  Avalia custo incremental c(e) para cada e ∈ C
4:  while ( C ≠ ∅) do
5:      c^min ← min {c(e) | e ∈ C}
6:      c^max ← max {c(e) | e ∈ C}
7:      RCL ← {e ∈ C | c(e) ≤ c^min + α(c^max − c^min)}
8:      Selecione um elemento s ∈ RCL aleatoriamente
9:      Solution ← Solution ∪ {s}
10:     Atualiza o conjunto candidato C
11:     Reavalia o custo incremental c(e) para todo e ∈ C
12: end while
13: return Solution
```

- C^{min} e C^{max}: menor e maior custo incremental, respectivamente;
- Quando $\alpha = 0$, construção é puramente gulosa;
- Quando $\alpha = 1$, construção é aleatória.

2.3.4. GRASP Paralelo

Heurísticas e metaheurísticas são no geral eficientes quanto ao tempo de execução quando comparadas com algoritmos exatos para resolver problemas de otimização combinatória. Porém quando utilizadas para instâncias de grande porte para tais problemas, o tempo computacional para explorar o espaço de busca na fase de busca local como por exemplo no GRASP, pode ser muito alto (ROCHA, 2008).

A estratégia usada para a implementação do GRASP paralelo aqui apresentado segue a abordagem de *Múltiplas Trajetórias Independentes.* Nesta abordagem a troca de informações é limitada a fase inicial e final encontrada por cada processo do sistema onde, inicialmente, um processo mestre faz a leitura dos dados do problema que são repassados para os demais processos restantes. Cada processo executa uma quantidade N de vezes, onde N é o número máximo de iterações do GRASP dividido pela quantidade de processos. Já, na fase final, o processo mestre recebe a melhor solução de cada processo, separando o melhor dentre eles como a solução final (ROCHA, 2008).

Vale ressaltar que, como o método metaheurístico GRASP utiliza de uma abordagem aleatorizada na sua fase de construção da solução inicial e busca local, é necessário o recebimento de uma semente (*Seed*) para a geração de valores aleatórios, que devem ser diferente para cada processo corrente do método, uma vez que, se dois processos ou mais receberem a mesma semente geradora de números aleatórios, a solução gerada na fase inicial será a mesma por todos esses processos.

2.4. GPU – Graphics Processing Unit

A Unidade de Processamento Gráfico, do inglês *Graphics Processing Unit (GPU),* é um microprocessador especializado em processamento de funções gráficas. É comumente encontrada em computadores pessoais, estações de trabalho, celulares e principalmente em consoles de videogame. Inicialmente foram projetadas como aceleradores gráficos, suportando somente *pipelines* de função fixa específicos. Mas atualmente tem sido também utilizada para processamento de dados gerais.

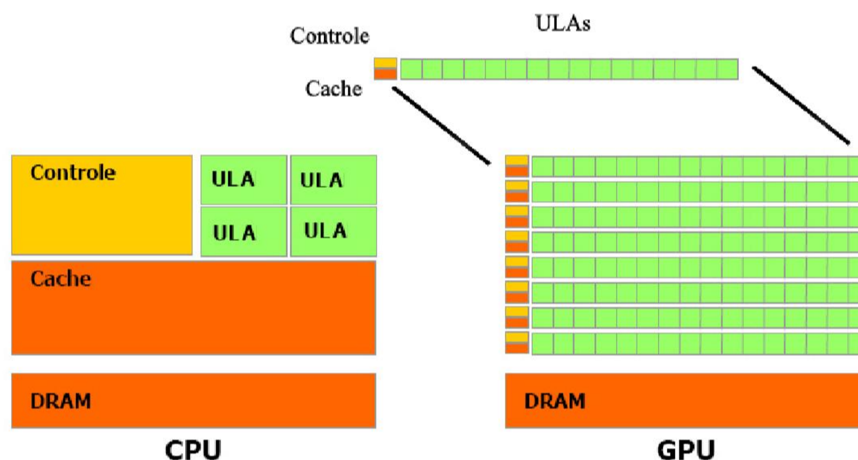
Operações gráficas como transformações geométricas, aplicação de iluminação, processamento de vértices e fragmentos dentre outras são operações em que não há dependência de dados a serem processados, com isso, vários cálculos podem ser executados paralelamente sem que haja alterações no resultado final. Este fato fez com que a GPU se especializasse em barramentos, para transportar a grande quantidade de dados envolvida e em paralelismo para realizar os vários cálculos, o que explica a principal razão da tamanha discrepância no desempenho computacional entre GPU e CPU em aplicações desta natureza.

A arquitetura de uma CPU como ilustrado na figura 5, é otimizada para o desempenho de código sequencial. Ela utiliza uma lógica de controle sofisticada para permitir que instruções de uma única *thread* de execução sejam executadas em paralelo ou mesmo fora de sua ordem sequencial, enquanto mantém a aparência de execução

sequencial. Grandes memórias cache são fornecidas para reduzir as latências de acesso a instruções e dados de aplicações grandes e complexas. Nem a lógica de controle nem as memórias cache contribuem para a velocidade máxima de cálculo (DAVID, WEN-MEI, 2011).

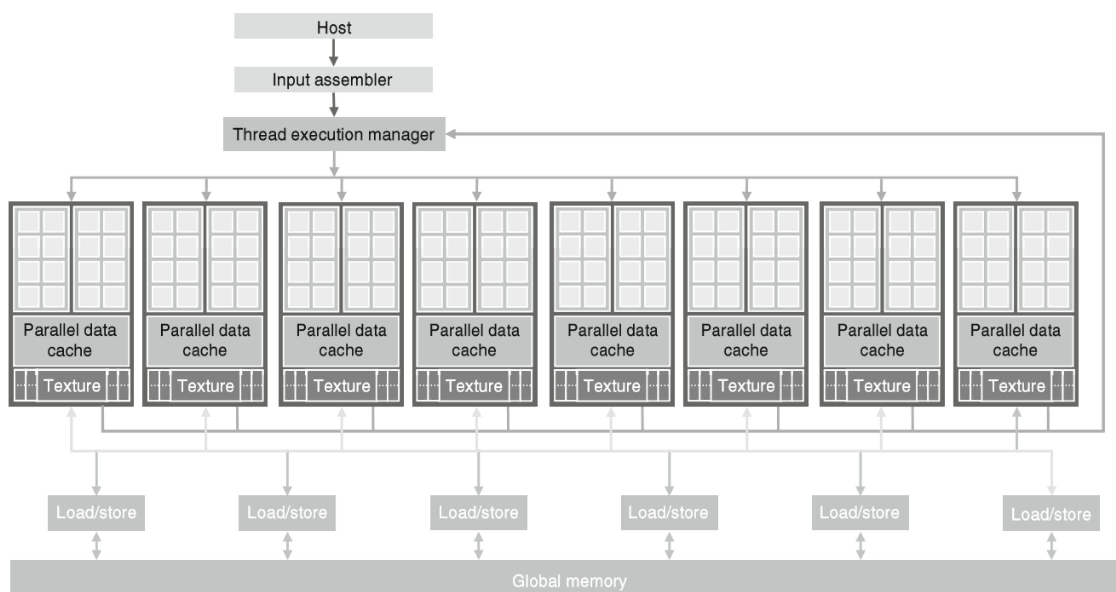
A arquitetura da GPU (vide figura 5) procura maximizar a área do *chip*, o consumo de energia dedicado aos cálculos de ponto flutuante e otimizar a vazão de execução do número maciço de *threads*. O *hardware* tira proveito de um grande número de *threads* de execução para encontrar trabalho para fazer, enquanto algumas delas estão esperando por acesso a memória de longa latência, minimizando assim a lógica de controle exigida para cada *thread* de execução. Pequenas memórias cache são fornecidas para ajudar a controlar os requisitos de largura de banda dessas aplicações, de modo que múltiplas *threads* acessando os mesmos dados da memória não precisem ir todas até a DRAM (DAVID, WEN-MEI, 2011).

Figura 5 - Arquitetura CPU vs GPU (ALVES, 2014)



A figura 6 mostra a arquitetura de uma GPU típica e preparada para a plataforma CUDA que será explicada adiante. Ela é organizada em uma matriz de multiprocessadores de *streaming* (SMs). Na figura 6, dois SMs formam um bloco, logo temos dezesseis SMs e oito blocos, no entanto este número pode variar de uma geração de GPUs para outra. Cada SM tem um certo número de processadores de *streaming* (SPs) que compartilham a lógica de controle e a cache de instruções. Cada GPU vem com uma memória global para escrita e leitura tanto pela GPU quanto CPU e apresenta uma alta latência de acesso (DAVID, WEN-MEI, 2011).

Figura 6 - Arquitetura típica de uma GPU CUDA (DAVID, WEN-MEI, 2011)



Usando linguagens de alto nível, aplicativos acelerados por GPU executam as partes sequenciais de suas cargas de trabalho na CPU – que é otimizada para desempenho com um único segmento (*thread*) – ao mesmo tempo em que aceleram o processamento em paralelo da GPU. Isso é chamado de "computação com GPU" (NVIDIA, 2018).

A computação com GPU é possível porque as atuais GPUs fazem muito mais do que processar imagens: lidam com um *teraflop* de desempenho de ponto flutuante e processam tarefas de aplicativos projetados para tudo, desde finanças, até medicina (NVIDIA, 2018).

Programas em GPU tem sido criados com a utilização de plataformas próprias a este tipo de programação. Estas plataformas são de mais fácil entendimento e associação se comparadas às API gráficas por abstraírem questões relacionadas ao processamento gráfico. Entretanto, sua implementação ainda exige um maior esforço do programador, já que necessita de um entendimento quanto a arquitetura da GPU e a forma como estes programas são executados. Um exemplo de uma plataforma é a criação da NVIDIA, a CUDA, que será apresentado em detalhes na próxima seção.

2.5. CUDA

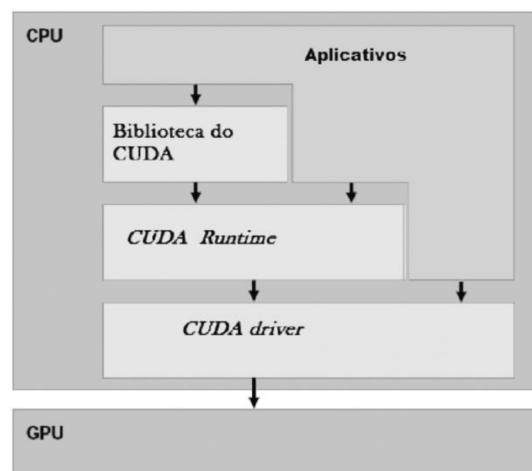
A CUDA acrônimo advindo do inglês *Compute Unified Device Architecture* é uma plataforma de computação paralela e uma arquitetura de abstração, com um modelo de programação embutido desenvolvido pela NVIDIA para computação de propósito geral em unidades de processamento gráfico (GPUs) (P. ROCHA; F FILHO, 2010) (NVIDIA, 2018). O modelo de programação CUDA utiliza as bibliotecas populares para desenvolvimento como C, C++, Fortran, Python e MATLAB, expressando paralelismo por meio de extensões na forma de palavras-chave básicas, sendo necessário possuir uma

placa de vídeo da NVIDIA para executar programas que utilizem tais bibliotecas. Recomenda-se a utilização de CUDA quando uma peque parte do processamento pode ser replicada e executada paralelamente em diversos núcleos - isso porque os núcleos da GPU são menos poderosos, porém mais numerosos (NVIDIA, 2018).

2.5.1. API CUDA

CUDA é baseado em componentes de *software* e *hardware*, onde a comunicação entre a CPU e GPU necessita de uma pilha de *softwares* que determina a hierarquia de compilação, conhecida como API CUDA. A utilização de uma pilha de *software* viabiliza a abstração no desenvolvimento de aplicações utilizando a plataforma. Desta forma a complexidade da comunicação entre CPU e GPU fica encapsulada na API. O modelo proposto visualiza a GPU como co-processador que é chamado de *Device* e a CPU como responsável pelo processamento principal, chamada de *Host*. Na figura 7 são representadas as camadas dessa pilha: Biblioteca da CUDA, CUDA *Runtime*, CUDA *Driver*. A camada de aplicação refere-se aos aplicativos que utilizam o modelo de programação CUDA que pode se comunicar com todos os níveis da pilha direta ou indiretamente (P. ROCHA; F FILHO, 2010).

Figura 7 - Camadas da API CUDA



As bibliotecas CUDA são implementações para dar suporte a NVIDIA CUDA *Driver*, disponibilizando o acesso aos recursos computacionais das GPUs. Apresentam funções auxiliares para criar e destruir objetos de vetores e matrizes no espaço de memória da GPU, sendo estes utilizados para a gravação e recuperação de dados que serão transferidos para a CPU. A biblioteca é autossuficiente no seu nível da pilha, isto é, não é necessária nenhuma interação direta com a CUDA *Driver* (P. ROCHA; F FILHO, 2010).

A CUDA *Runtime* API atua como uma intermediária entre o desenvolvedor e o D*river*, de forma a facilitar a programação mascarando alguns detalhes de baixo nível. A biblioteca *Runtime* é dividida em componentes que gerenciam operações do *host*, do *device* e componentes comuns, como tipos adicionais e subconjuntos de funções da biblioteca padrão C que podem ser utilizados no *host* quanto no *device*. A API *Runtime* é responsável por realizar implicitamente a inicialização, gerenciamento de contexto e dos módulos (P. ROCHA; F FILHO, 2010).

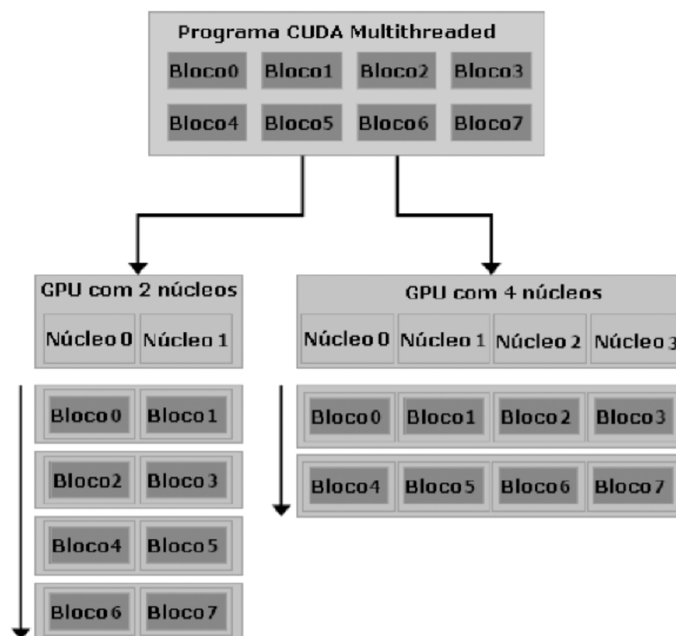
A CUDA *Driver* API corresponde a camada intermediária entre o código compilado e a GPU, sendo implementada na biblioteca dinâmica nvcuda. Por atuar no último nível de abstração, apresenta maior complexidade na sua manipulação direta pela necessidade de maior entendimento das funcionalidades de comunicação com a GPU utilizadas pela plataforma CUDA. Essa flexibilidade de implementação possibilita um controle maior dos recursos. Com isso o desenvolvedor pode realizar otimizações manuais de algumas funcionalidades presentes nessa camada, viabilizando o desenvolvimento de dispositivos mais complexos e específicos (P. ROCHA; F FILHO, 2010).

2.5.2. Modelo de Programação

A plataforma CUDA segue um modelo de programação escalável apresentando em seu núcleo três abstrações principais: hierarquia de grupos de *threads*, hierarquia de memória compartilhada e sincronização de barreira – que são expostas ao programador de forma simples para utilizar, por meio de um conjunto mínimo de extensão de linguagem. As abstrações proporcionam paralelismo de dados e de *threads*, particionando o problema em subproblemas que podem ser resolvidos de forma independente, em paralelo por blocos de *threads*. Ainda esses subproblemas podem ser particionados em partes menores sendo resolvidos de forma cooperativa em paralelo por *threads* dentro de cada um dos blocos formados. Esse tipo de decomposição permite que as *threads* cooperem para a resolução de cada subproblema (P. ROCHA; F FILHO, 2010) (ALVES, 2014).

Cada bloco de *threads* pode ser programado em qualquer um dos multiprocessadores disponíveis dentro de uma GPU, em qualquer ordem, simultaneamente ou sequencialmente, para que um programa CUDA compilado possa ser executado em qualquer número de multiprocessadores, conforme ilustrado na Figura 8, e somente em tempo de execução o sistema precisa conhecer a contagem de multiprocessadores físicos.

Figura 8 - Escalabilidade de um programa Multithread CUDA



Essa forma de estruturação da execução altamente paralelizada dos programas utilizando a arquitetura CUDA necessita da implementação de *Kernels*, Hierarquia de *Threads* e de Memória que em conjunto tornam a aplicação extremamente paralelizável influenciando diretamente no desempenho destas.

2.5.3. Kernels

Os códigos das aplicações a serem aceleradas utilizando a plataforma CUDA são fortemente modularizados. Isso significa que os programas apresentam suas funcionalidades implementadas em funções. No contexto CUDA a função executada na GPU que é chamada pela CPU (*host*) é denominada de *Kernel* (P. ROCHA; F FILHO, 2010).

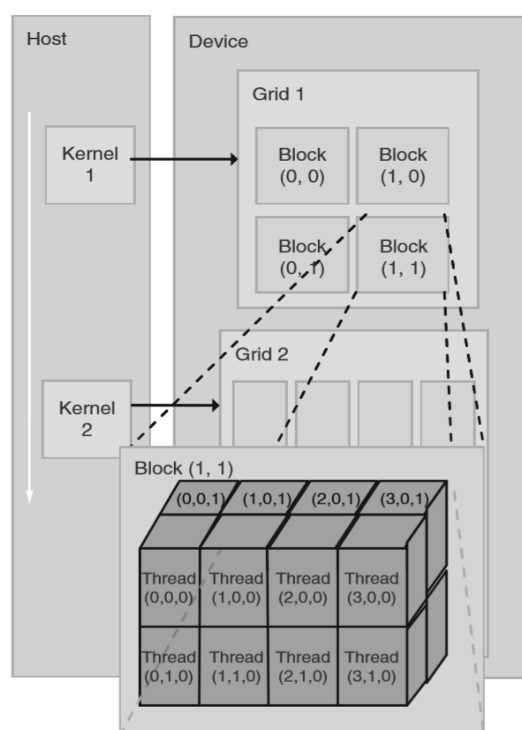
O *kernel* é mapeado por um conjunto de *threads*, sendo assim o código presente neste é executado por cada *thread* do conjunto. Desta forma, em cada chamada ao *Kernel* deve ser especificada sua configuração, com o número de blocos em cada *grid* e o número de *threads* em cada bloco, já a definição da quantidade de memória compartilhada a ser alocada é opcional. Quando é realizada a chamada ao *Kernel* o fluxo de execução sai da CPU (*host*) e passa para a GPU (*device*). Então o código presente no *Kernel* é executado *N* vezes em paralelo por *M* diferentes *threads* de CUDA, sendo a paralelização realizada de forma automática. Quando a execução paralela do *Kernel* termina o fluxo de execução volta para a CPU (*host*) retomando a execução do programa principal (P. ROCHA; F FILHO, 2010).

2.5.4. Hierarquia de Thread

A base da execução dos programas na plataforma CUDA são as *threads*. Para organizar a repartição dos dados entre as *threads*, bem como sua organização e distribuição ao *hardware*, a CUDA trabalha com uma Hierarquia de *Thread* (P. ROCHA; F FILHO, 2010).

Por conveniência, uma *thread* na CUDA é identificada por um vetor de três componentes, para que cada *thread* seja identificado como unidimensional, bidimensional ou tridimensional, formando um bloco de execução também de iguais dimensões. Um conjunto blocos formam um *grid*, uma estrutura completa de distribuição para a execução do k*ernel*.

Figura 9 - Organização da grade CUDA (DAVID, WEN-MEI, 2011)



A figura 9 mostra uma g*rid* 2D que consiste em quatro blocos organizados em uma matriz 2x2. Os blocos são organizados em matrizes 3D de *threads*. Todos os blocos em uma g*rid* têm as mesmas dimensões. Cada bloco é organizado em matrizes 4x2x2. Neste exemplo tem-se 4 blocos com 16 t*hreads* cada, com um total de 64 t*hreads* na *grid* (DAVID, WEN-MEI, 2011).

A organização das *threads* em blocos é necessária para que estas possam ser executadas de forma independente, garantindo a escalabilidade dos recursos disponíveis na GPU. Assim é possível executá-las em qualquer ordem, em paralelo ou em série. As *threads* dentro de um bloco cooperam com a partilha de dados através da memória

compartilhada e sincronizam sua execução para coordenar os acessos a memória (P. ROCHA; F FILHO, 2010).

Há um limite para o número de *thread* por bloco, uma vez que todas as *threads* de um bloco devem residir no mesmo núcleo do processador e devem compartilhar os mesmos recursos de memória limitados desse núcleo. Assim a quantidade total de *thread* designada a um bloco deve ser compatível com os recursos de memória disponíveis no núcleo de processamento da GPU, para que não ocorra a subutilização e nem extrapolação dos mesmos,

Nas GPUs atuais, um bloco de *thread* pode conter até 1024 *threads*. No entanto, um *kernel* pode ser executado por vários blocos de *thread* da mesma forma, de modo que o número total de *thread* seja igual ao número de *thread* por bloco multiplicado pelo número de bloco.

Encadeamentos dentro de um bloco podem cooperar compartilhando dados através de alguma memória compartilhada e sincronizando sua execução para coordenar os acessos à memória.

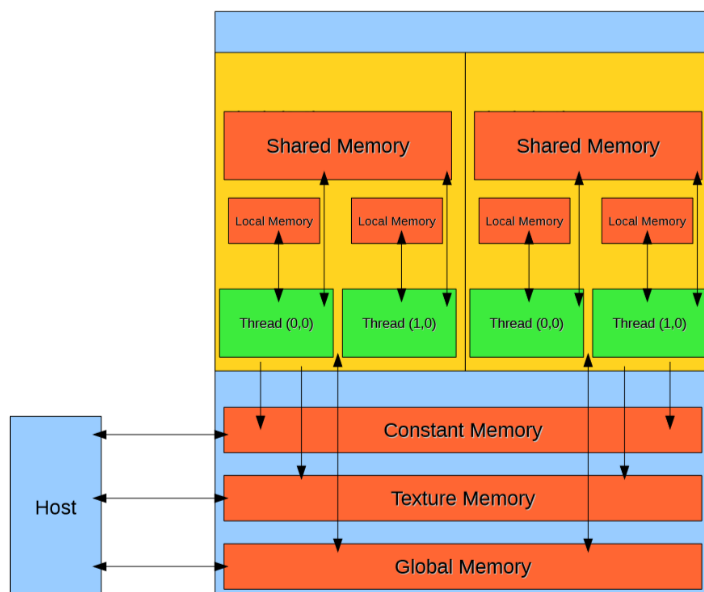
Um programa CUDA pode ser composto por centenas de *threads*, contudo nem todas estas *threads* executam simultaneamente com paralelismo real. Para gerenciar a execução destas *threads* existe um escalonador implementado via *hardware* em cada multiprocessador da GPU, cada um desses multiprocessadores é responsável por cuidar de um agrupamento de 32 *threads* conhecido como *warp*. O agrupamento e gerenciamento das *threads* é realizado pela arquitetura *Single Instruction Multiple Thread* (SIMT) ou uma instrução para múltiplos processos. Quando uma g*rid* ou bloco de *thread* for executado por um multiprocessador, são divididos em *warp* que são escalonados pelo SIMT (P. ROCHA; F FILHO, 2010).

2.5.5. Hierarquia de Memória

Cada *thread* do programa tem uma memória local, acessada única e exclusivamente por ela, para o armazenamento dos dados. Cada bloco acessa uma área específica da memória compartilhada. Desta forma todas as *threads* pertencentes a um mesmo bloco podem acessar esse espaço de memória e compartilhar dados entre si. As *grids* tem acesso a memória global, que é a memória principal da GPU, desta forma todas as *threads* que compõem o programa podem compartilhar os dados processados. Assim *threads* de blocos e g*rids* diferentes podem compartilhar dados umas com as outras (vide Figura 10) (P. ROCHA; F FILHO, 2010).

Ainda existem outros dois espaços de memória somente de leitura, textura e constante, que são otimizados para diversos usos da memória. A memória Global, Textura e Constante são persistentes, não deixando de existir quando a execução do k*ernel* termina. A memória de Textura oferece diferentes modos de endereçamento, bem como dados de filtragem para alguns formatos de dados específicos (vide Figura 10) (ALVES, 2014).

Figura 10 - Distribuição de Memória GPU



A abordagem CUDA é como a GPU, altamente paralela, porém ela divide o conjunto de dados em pedaços menores na memória *on-chip*. Esta memória apresenta os tipos: *Cache* Constante para acesso à memória constante, *Cache Texture* para acesso à memória de textura, conjunto de registradores e memória compartilhada. Assim as memórias constante e de textura possuem melhor desempenho do que a local e global devido a utilização de *cache* no acesso aos dados. Já as memórias compartilhadas e registradores tem o melhor desempenho entre todos os tipos por estarem no mesmo *chip* do microprocessador, sendo a memória global a mais custosa no acesso, por estar mais distante fisicamente do processador (P. ROCHA; F FILHO, 2010).

Todo o gerenciamento de memória passa pelo *runtime* da CUDA, incluindo a alocação, desalocação e transferência de dados entre as memórias do *host* (CPU) e do *device* (GPU). As diretivas responsáveis por esse gerenciamento estão disponíveis na biblioteca CUDA (P. ROCHA; F FILHO, 2010).

2.6. CUDA em C

A plataforma CUDA é utilizada comumente para solucionar problemas complexos utilizando a GPU, tornando a programação neste modelo algo não tão simples quanto a programação procedural. Com o objetivo de tornar mais rápida a curva de aprendizagem dos programadores, a plataforma da NVIDIA apresenta suporte a várias linguagens e interfaces de programação como C/C++, Fortran, OpenCL e DirectCompute (P. ROCHA; F FILHO, 2010).

2.6.1. Compilação com NVCC

A NVIDIA disponibiliza o compilador NVCC para a compilação dos programas CUDA, de modo a facilitar o processo de compilação ao fornecer opções de linhas de comando simples e familiares e invoca diversas ferramentas para cada estágio de compilação (P. ROCHA; F FILHO, 2010).

O primeiro estágio de compilação é realizar a separação do código da CPU do código da GPU. Para realizar esta divisão é usado um processador no *front-end*, o *Edison Design Group* (EDG). Este realiza a análise do código produzindo duas saídas: código C puro para o *host* (CPU) e um objeto cubin (formato binário) para o *device* (GPU). O próximo passo é realizar a compilação dos códigos. O código C puro é compilado pelo compilador nativo (GCC, *Microsoft Compiler*, dentre outros) e o código da GPU é compilado com uma versão customizada do Open64. Este é um compilador de código aberto que gera código para execução em paralelo. Após a compilação pelo Open64 é gerado um código no formato *assembly* chamado PTX. Este código é convertido para código executável em tempo de execução (*Just-in-time*). Assim é utilizado um tradutor de *assembly* que gera um código, específico para determinado modelo de GPU, utilizando o código PTX (P. ROCHA; F FILHO, 2010).

A compilação *Just-in-time* do código ocasiona o aumento do tempo de carregamento do aplicativo CUDA, mas permite que os aplicativos possam tirar proveito das melhorias mais recentes do compilador. Além de ser a única forma das aplicações rodarem em dispositivos que não existiam no momento, no qual, as aplicações foram précompiladas (P. ROCHA; F FILHO, 2010).

2.6.2. Fluxo de Execução

O fluxo de execução define os passos a serem implementados para que a GPU funcione corretamente em conjunto com a CPU. Primeiramente, os dados a serem processados na GPU devem ser devidamente copiados para a memória principal da GPU. Em seguida, deve ser feito a configuração de execução da GPU, definindo a quantidade de *threads* e blocos bem como a sua dimensão, por meio da chama de k*ernel*. Nesse momento, os dados são processados em paralelo em cada núcleo disponível na GPU segundo a configuração feita. Logo após ao fim deste processamento, os dados resultantes do processamento na GPU devem voltar para a CPU por meio de cópia, para futuros processamentos.

Desta forma, o fluxo de execução de uma aplicação que utiliza a plataforma CUDA altera sua execução da CPU para a GPU e depois retorna a CPU.

2.6.3. Programando com CUDA em C

Para o desenvolvimento de aplicações em C que utilizam a plataforma CUDA, é necessário aplicar alguns conceitos utilizados como qualificadores, variáveis, configuração de execução e função de sincronização.

Os qualificadores na plataforma CUDA são responsáveis por designar o local de execução das funções *host* (CPU) e *device* (GPU) e também a localização de variáveis nos dispositivos referidos.

Os qualificadores **__device__**, usado na declaração de funções que são executados somente na GPU e só podem ser chamadas a partir do mesmo, **__host__**, usado na declaração de uma função que será executada na CPU e só podem ser chamadas a partir do mesmo e o qualificador **__global__** que é usado na definição de uma função do *Kernel*, função onde a chamada ocorre no *host* e sua execução acontece no *device*, são qualificadores que designam o local de execução das funções (P. ROCHA; F FILHO, 2010).

Já os qualificadores que implicam na localização de variáveis são **__shared__**, que define uma variável compartilhada a todas as *threads* pertencentes ao mesmo bloco, **__device__**, define variáveis que residem no device na memória global e **__constant__** que declara variáveis que residem num espaço de memória constante tendo a mesma acessibilidade de uma variável **__device__** (P. ROCHA; F FILHO, 2010).

Funções declaradas com o qualificador **__global__** representam o *Kernel*. São funções chamadas pelo *host* e executadas no *device* que devem ter alguns parâmetros que constituem as configurações de execução. Na API *Runtime* CUDA a configuração é expressa da seguinte forma:

<<< **Dg, Db, NS, S** >>>

Onde:

Dg: Variável do tipo dim3 e especifica a dimensão e o tamanho da *grid*;

Db: Variável do tipo dim3 e especifica a dimensão e o tamanho de cada bloco;

NS: Variável do tipo **size_t** e especifica o tamanho, em *bytes*, do espaço que será alocado dinamicamente na memória compartilhada por bloco;

S: Variável do tipo **cudaStream_t** e especifica um stream adicional.

Nas chamadas ao k*ernel* não é necessário realizar a configuração completa apresentada. Os parâmetros **NS** e **S** são opcionais. Para exemplificar, no trecho de código da figura 11 é representada a chamada do k*ernel* no *host* (CPU) e a assinatura da mesma, utilizando a configuração incompleta. As variáveis dentro dos parênteses correspondem

aos parâmetros do *kernel*. A variável *blocks* e *threads* representam o número de blocos e *threads* por bloco respectivamente que a GPU irá executar.

Figura 11 - Chamada do Kernel e Assinatura da função

```
// Chamada de Kernel pela CPU.
//Executa a funcao `parallelGRASP` no device
parallelGRASP << <blocks, threads>> >(max_iter, n_itens, capacidade_mochila, tamanho_RCL, seed);


//GRASP
//Assinatura da funcao no device
__global__ void parallelGRASP(int max_iter, int n_itens, int capacidade_mochila, int tamanho_RCL, int seed);
```

3. Metodologia

Neste capítulo são apresentados os materiais e métodos utilizados para o desenvolvimento desse projeto, bem como bibliotecas C/C++ utilizadas, as instâncias de teste utilizadas, abordagem de geração da solução inicial e busca local da metaheurística GRASP desenvolvida e também a paralelização do código utilizando GPU.

3.1. Materiais

Todo trabalho foi desenvolvido em uma máquina com a seguinte configuração:

- Processador Intel i5 – 4200U 1.60GHz;
- 6GB de memória RAM;
- Placa de vídeo NVIDIA GeForce GT 740m com 384 CUDA core;
- Sistema operacional Windows 10 Pro versão 1709 64-bit
- CUDA *Toolkit* versão 8.0

A Figura 12 apresenta em detalhes as propriedades da GPU.

Figura 12 - Propriedades da GPU

Attributes	
Compute Capability	3.5
Maximums	
Threads per Block	1024
Threads per Multiprocessor	2048
Shared Memory per Block	48 KiB
Shared Memory per Multiprocessor	48 KiB
Registers per Block	65536
Registers per Multiprocessor	65536
Grid Dimensions	[2147483647, 65535, 65535]
Block Dimensions	[1024, 1024, 64]
Warps per Multiprocessor	64
Blocks per Multiprocessor	16
Single Precision FLOP/s	792,96 GigaFLOP/s
Double Precision FLOP/s	33,04 GigaFLOP/s
Multiprocessor	
Multiprocessors	2
Clock Rate	1,032 GHz
Concurrent Kernel	true
Max IPC	7
Threads per Warp	32
Memory	
Global Memory Bandwidth	14,4 GB/s
Global Memory Size	2 GiB
Constant Memory Size	64 KiB
L2 Cache Size	512 KiB
Memcpy Engines	1
PCIe	
Generation	2
Link Rate	5 Gbit/s
Link Width	4
Environment	

Todo código desenvolvido foi escrito usando a linguagem de programação C/C++ utilizando para tal a IDE Visual Studio 2015 Enterprise, um ambiente de desenvolvimento integrado da Microsoft, capaz de oferecer suporte completo para a criação de *softwares* para Windows, Mac e Linux, além de estrutura para desenvolvimento *web* e de aplicativos para Android e iOS e também suporte a plataforma CUDA.

O conjunto de testes usado foi um *dataset* disponibilizado pelo professor *David Pisinger* (PISINGER, 2005), que contém instâncias de tamanhos variados nos quais os testados neste trabalho foram de 100, 500, 1000 e 2000 itens. Cada arquivo de entrada contém em sua primeira linha o número de itens seguido da restrição de peso da mochila, e em cada linha sucessivamente contém o valor e peso respectivamente de cada item, sendo um item dado por linha. Os arquivos usados foram:

Tabela 1 - Instâncias de teste.

Nome	Itens	Capacidade	Ótimo
knapPI_1_100_1000_1	100	995	9147
knapPI_1_500_1000_1	500	2543	28857
knapPI_1_1000_1000_1	1000	5002	54503
knapPI_1_2000_1000_1	2000	10011	110625

Para o desenvolvimento da metaheurística GRASP paralelo sendo executado somente na CPU foi utilizado a biblioteca *Pthreads* da norma Posix, uma biblioteca para a linguagem C que permite a execução do mesmo projeto usando diferentes *threads*.

3.2. Métodos

Para solucionar o Problema da Mochila, um problema de otimização combinatória e avaliar o desempenho da plataforma CUDA, implementou-se a Metaheurística GRASP de forma sequencial e paralelo em CPU e paralelo em GPU, com função objetivo de maximizar o valor do ganho do subconjunto de itens escolhidos para incorporar a solução sem exceder a capacidade máxima de alocação deste conjunto.

O algoritmo inicia-se recebendo os dados referentes ao problema da mochila como quantidade de itens, capacidade máxima da mochila, peso e valor referente a cada item que englobam o conjunto do problema, por meio de um arquivo texto. Após a leitura desse arquivo, os dados são salvos em uma estrutura de dados criada para facilitar o armazenamento dos mesmos. Os demais dados como número de iterações, quantidade de *thread* e blocos, tamanho da RCL e nome da instância a ser executada. Estes são definidos e passados por parâmetro dentro do algoritmo.

A estrutura de dados usada para guardar os dados referente ao problema, especificamente o ganho (valor/peso), peso e valor de cada item é um vetor com N posições, sendo N igual ao número de itens. Cada posição guarda um tipo de dado intitulado como *item* que contém peso, valor e ganho. Para armazenar os resultados parciais de cada *thread* e a solução final, usou-se um vetor também de tamanho N do tipo booleano, logo se o valor na posição 10 do vetor resultado for 1, indica que o item 10 incorpora o conjunto resultado, se for igual a 0 o item não pertence a solução.

Tabela 2 – Exemplo da estrutura de dados 'item' preenchida com 5 posições.

Peso	Valor	Ganho
9	7	0.77
8	6	0.75
5	9	1.8
2	8	4
4	3	0.75

Antes de iniciar os passos de *facto* do GRASP, é executado um algoritmo de ordenação na instância corrente, ordenando de forma decrescente os itens segundo o seu índice de ganho. Esse índice é calculado item por item, dividindo o valor do item pelo seu peso (vide Tabela 2). Este pré-processamento busca minimizar as iterações na busca por um elemento segundo seu índice de ganho, uma vez que os melhores itens estarão sempre nas primeiras posições.

As implementações foram projetadas seguindo os passos básicos do GRASP como descrito anteriormente na seção 2.3 e mostrado na Figura 13: Construir a solução inicial

aleatória. Executar a busca local com objetivo de melhorar a solução inicial encontrada. Se a solução corrente for melhor que a solução da iteração passada, a solução final é atualizada, recebendo sempre a melhor solução já encontrada. Todos os passos se repetem um número máximo de vezes informado como parâmetro do GRASP.

Figura 13 - Passos Básicos do GRASP

```
void simpleGRASP
(int max_iter, int quantidade_itens, int bin_capacity, item *itens, bool *solucao_final, int tamanho_RCL, int seed, int &max_valor) {

    //solução parcial da iteração
    bool *solucao_parcial;
    solucao_parcial = (bool *)malloc(quantidade_itens * sizeof(bool));
    //inicializando - todos itens fora da solução == 0
    for (int i = 0; i < quantidade_itens; i++) {
        solucao_parcial[i] = 0;
    }
    int  i, j = 0;
    int valor_parcial = 0, peso_parcial = 0;
    for (i = 0; i < max_iter; i++) {
        /*inicio grasp*/
        //gera solução inicial
        GreedyRandomizedConstruction(quantidade_itens, solucao_parcial, itens, bin_capacity, seed + i, tamanho_RCL, peso_parcial, valor_parcial);
        //faz a busca local tentando melhorar a solução gerada
        LocalSearch(quantidade_itens, solucao_parcial, bin_capacity, itens, seed + i, valor_parcial, peso_parcial, tamanho_RCL);
        //atualiza a solução
        if (valor_parcial > max_valor) {
            max_valor = valor_parcial;
            UpdateSolution(solucao_parcial, solucao_final, quantidade_itens);
        }
        //a cada iteração reseta peso, valor e a solução parcial
        peso_parcial = 0;
        valor_parcial = 0;
        for (j = 0; j < quantidade_itens; j++) {
            solucao_parcial[j] = 0;
        }
    }
    free(solucao_parcial);
```

Ao iniciar o GRASP em sua fase de construção da solução inicial, faz-se a construção da lista de candidatos restritos (RCL) com base nos elementos que podem incorporar a solução inicial. Esses elementos são os melhores segundo seu índice de ganho que foi calculado anteriormente. Com a RCL criada, é feito um sorteio de um item da RCL para incorporar a solução. Esse processo se repete até que nenhum item possa incorporar a RCL e consequentemente a solução inicial. As figuras 14 e 15 ilustram esses passos.

Figura 14 - Chamada da função que cria a solução inicial

```
void GreedyRandomizedConstruction
(int quantidade_itens, bool *solucao_parcial, item *itens, int capacidade_mochila, int seed, int tamanho_RCL, int &peso, int &valor) {

    //completa com os melhor elementos segundo seus indeces
    max_indice(quantidade_itens, solucao_parcial, itens, capacidade_mochila, tamanho_RCL, peso, valor, seed);
}
```

Figura 15 - Construção da solução inicial

```
//completa a solução com os melhores itens segundo seus indices - randomico segundo o tamanho_RCL
void max_indice(int quantidade_itens, bool *solucao_parcial, item *itens, int capacidade_mochila, int tamanho_RCL, int &peso, int &valor, int seed) {
    //variavel que ira receber um valor randomico
    int idRand = 0;
    //peso que ja foi alocado na mochila
    int capacidade = peso;
    //instanciando a RCL
    int *rcl = (int *)malloc(tamanho_RCL * sizeof(int));
    int i = 0;
    //itens na RCL
    int cont_rcl = 1;
    while (cont_rcl > 0) {
        cont_rcl = 0;
        for (i = 0; i < quantidade_itens; i++) {
            if (solucao_parcial[i])
                continue;
            if (itens[i].peso + capacidade <= capacidade_mochila) {
                rcl[cont_rcl] = i;
                cont_rcl++;
            }
            if (cont_rcl == tamanho_RCL) {
                break;
            }
        }
        if (cont_rcl > 0) {
            idRand = rand() % cont_rcl;
            solucao_parcial[rcl[idRand]] = 1;
            capacidade += itens[rcl[idRand]].peso;
            valor += itens[rcl[idRand]].valor;
            peso += itens[rcl[idRand]].peso;
        }
    }
    free(rcl);
```

Na fase de busca local (vide Figura 16), é feita a tentativa de achar no espaço de busca uma solução vizinha melhor que a solução corrente. Para gerar um vizinho usa-se a estratégia de alterar um elemento por vez. O elemento a ser alterado é selecionado por meio do sorteio usando o método Roleta (Vide Figura 17), onde cada item presente na solução tem uma fatia na roleta de tamanho inversamente proporcional ao seu ganho, logo os itens com menor índice de ganho serão os elementos que terão a maior fatia na roleta, ocasionando a maior probabilidade de ser sorteado para deixar de fazer parte da solução. Este elemento sorteado volta para o espaço de busca. Feito isso, um novo elemento é sorteado usando a mesma técnica para gerar a solução inicial, para incorporar a solução corrente, podendo o item que foi retirado de a solução voltar sendo este o critério de parada do método. Enquanto a solução vizinha gerada for melhor, os passos se repetem.

Figura 16 - Busca Local GRASP

```
//Busca Local
void LocalSearch
(int quantidade_itens, bool *solucao_parcial, int capacidade_mochila,
    item *itens, int seed, int &valor_parcial, int &peso_parcial, int tamanho_RCL) {

   int valor_parcial_busca = valor_parcial;
   bool flag = true;
   while (flag) {
       flag = false;
       //Sorteia um numero pra sair da solucao
       roleta(solucao_parcial, itens, quantidade_itens, peso_parcial, valor_parcial);
       //completa a solucao
       max_indice(quantidade_itens, solucao_parcial, itens, capacidade_mochila, 1, peso_parcial, valor_parcial, seed);
       //se a solucao for melhor que a passada continua, se nao sai
       if (valor_parcial > valor_parcial_busca) {
           valor_parcial_busca = valor_parcial;
           flag = true;
       }
   }
}
```

Figura 17 -Método Roleta

```
void roleta(bool *solucao_parcial, item *itens, int quantidade_itens, int &peso, int &valor) {
    struct roleta {
        int id;
        float ganho;
    };
    float soma_ganhos = 0; int tamanho = 0;
    for (int i = 0; i < quantidade_itens; i++) {
        if (solucao_parcial[i]) {
            ++tamanho;
            soma_ganhos += itens[i].ganho;
        }
    }
    roleta *iten_roleta;
    iten_roleta = (roleta *)malloc(tamanho * sizeof(roleta));
    roleta aux;
    aux.ganho = 0; aux.id = 0;
    float anterior = 0;
    for (int j = 0; j < tamanho; j++) {
        for (int i = 0; i < quantidade_itens; i++) {
            if (solucao_parcial[i]) {
                if (itens[i].ganho > aux.ganho) {
                    aux.ganho = itens[i].ganho;
                    aux.id = i;
                }
            }
        }
        iten_roleta[j].ganho = (soma_ganhos - aux.ganho) + anterior;
        iten_roleta[j].id = aux.id;
        anterior += soma_ganhos - aux.ganho;
    }
    float id_rand = rand() % (int)anterior;
    int aux_id = 0;
    for (int i = 0; i < tamanho; i++) {
        if (iten_roleta[i].ganho >= id_rand) {
            aux_id = iten_roleta[i].id;
            break;
        }
    }
    solucao_parcial[aux_id] = 0;
    valor -= itens[aux_id].valor;
    peso -= itens[aux_id].peso;
}
```

Todos os passos apresentados até aqui, são comuns em todas as implementações feitas do GRASP neste trabalho, tanto em sua versão sequencial quanto nas suas versões

paralelas rodando em CPU ou GPU. As alterações que se fazem necessárias para executar o GRASP de forma paralela se dá na chamada inicial do GRASP. Onde tem-se o chamado em que um só processo executa todas as iterações na sua versão sequencial, é modificado para uma chamada de *kernel* no caso da GPU ou a criação de **pthreads** e definição das funções para cada uma rodar na CPU, como ilustrado nas figuras 18 e 19 que mostram a criação e execução das pthreads.

Figura 18 - Criação das Pthreads

```
for (int i = 0; i < threads; i++) {
    arg[i] = { max_iter, quantidade_itens, capacidade_mochila, itens, solucoes, tamanho_RCL, seed, i };
    //printf("In main: creating thread %d\n", i);
    int result_code = pthread_create(&thread[i], NULL, GRASP_thread, (void *)&arg[i]);
    assert(0 == result_code);
}

for (int i = 0; i < threads; i++) {
    int result_code = pthread_join(thread[i], NULL);
    assert(0 == result_code);
}
```

Figura 19 - Iniciando cada pthread para rodar o GRASP na CPU

```
typedef struct {
    int max_iter;
    int quantidade_itens;
    int capacidade_mochila;
    item *itens;
    bool *solucao;
    int tamanho_RCL;
    int seed;
    int id_thread;
}GRASP_starter;

void* GRASP_thread(void* init)
{
    GRASP_starter arg = *((GRASP_starter* )init);

    parallelGRASP((arg).max_iter, (arg).quantidade_itens, (arg).capacidade_mochila, (arg).itens, (arg).solucao, (arg).tamanho_RCL, (arg).seed, (arg).id_thread);

    return NULL;
}
```

Já na Figura 20 mostra a função *helper_cuda* onde é feito toda a configuração para alterar o fluxo de execução, ou seja, mudar o processamento do algoritmo da CPU para a GPU. A função ***cudaMemcpToSymbol()*** copia dados para a memória constante da GPU, ***cudaMalloc()*** aloca na memória global da GPU em espaço de memória e a função ***cudaMemcpy()*** copia dados para um espaço de memória alocado na memória global da GPU, ou da GPU para a CPU segundo o último parâmetro da função, ***cudaMemcpyHostToDevice*** CPU para GPU ou ***cudaMemcpyDeviceToHost*** GPU para CPU.

Como a chamada do kernel é assíncrono, o fluxo de execução também continua na CPU, logo para tal algoritmo é necessário fazer um sincronismo dos fluxos GPU e

CPU. O sincronismo fica por conta da função ***cudaDeviceSynchronize(),*** responsável por pausar o fluxo na CPU até que a GPU termina o processamento dos dados.

Figura 20 - Helper CUDA

```
//Helpre CUDA
//Aloca e copia todos dados pertinentes para processamento
void parallel_GRASP
(int max_iter, int quantidade_itens, int capacidade_mochila, item *itens, bool *solutions, int threads, int blocks, int tamanho_RCL, int seed) {

    //Aloca copia para memoria constante os itens
    cudaMemcpyToSymbol(constant_itens, itens, quantidade_itens * sizeof(item));

    //aloca na memoria global a solucao final de cada thread e as parciais de cada execucao
    bool* dev_soluctions;
    cudaMalloc((void**)&dev_soluctions, quantidade_itens * threads * blocks * sizeof(bool));

    bool* dev_soluctions_parcial;
    cudaMalloc((void**)&dev_soluctions_parcial, quantidade_itens * threads * blocks * sizeof(bool));

    //copia para a memoria global setando as solucoes como vazias (igual a 0)
    cudaMemcpy(dev_soluctions, solutions, quantidade_itens * blocks * threads * sizeof(bool), cudaMemcpyHostToDevice);
    cudaMemcpy(dev_soluctions_parcial, solutions, quantidade_itens * blocks * threads * sizeof(bool), cudaMemcpyHostToDevice);

    //Executando o algoritmo na GPU
    //Chamada de Kernel
    parallelGRASP << <blocks, threads>> >(max_iter, quantidade_itens, capacidade_mochila, dev_soluctions, dev_soluctions_parcial, tamanho_RCL, seed);

    //Esperando todas as threads executarem
    cudaDeviceSynchronize();

    //Copiando os resultados para a CPU
    cudaMemcpy(solutions, dev_soluctions, quantidade_itens * blocks * threads * sizeof(bool), cudaMemcpyDeviceToHost);

    cudaFree(dev_soluctions);
    cudaFree(dev_soluctions_parcial);
}
```

Antes de mudar o fluxo de execução para a GPU, todas as configurações já foram feitas. Todos os dados que serão processados na GPU já foram devidamente copiados para a memória da GPU e a quantidade de *threads* e blocos já foram definidos. Após a execução, os resultados obtidos na GPU são copiados para a memória RAM para processamentos futuros.

Os dados do problema referentes aos itens como peso, valor e índice de ganho são copiados para a memória constante da GPU, uma memória que só permite operação de leitura por funções do *device* e tem a latência de acesso menor que a memória global.

Nas abordagens paralelas do GRASP, durante a execução da metaheurística, cada *thread* salva sua melhor solução em um vetor. Ao final, quando todas as *threads* já terminaram seus processos, tem-se um vetor contendo todas as melhores soluções de cada *thread*, basta então executar uma iteração de complexidade $O(n)$ no vetor para encontrar a melhor solução e tornar ela a solução final.

Essa abordagem foi necessária para evitar o uso de semáforos ou algum tipo de controle de acesso a memória, criando uma área crítica na execução do algoritmo paralelo. Isso acarretaria numa lentidão no tempo de execução do mesmo, uma vez que toda *thread* ao final do processo, teria que ter acesso exclusivo ao vetor que armazena a solução final.

4. Experimentos Computacionais

Este capítulo apresenta os resultados obtidos nos experimentos realizados para a validação do método proposto GRASP, com suas variações sequencial, uma paralela em CPU, e paralela usando GPU. No caso das abordagens paralelas, é apresentada ainda uma comparação do método variando a quantidade de *threads*, blocos e número máximo de iterações.

Para obtenção dos resultados os experimentos realizados tanto com GRASP sequencial quanto com GRASP paralelo em CPU ou GPU foram repetidos 100 vezes. Isto é importante para analisar o comportamento (robustez) das técnicas, já que a metaheurística GRASP possui um componente probabilístico. Conjuntamente, em todos os experimentos, o número de iterações utilizada no GRASP foi determinado empiricamente para o valor 1.024, resultando numa boa relação entre o esforço computacional e a qualidade das soluções obtidas.

Os procedimentos serão avaliados quanto à média do tempo de execução e a qualidade de seus resultados. Para isso, cada experimento foi realizado sobre 4 instâncias de tamanho 100, 500, 1000 e 2000 itens, utilizando um computador pessoal como descrito na seção 3.1.

O parâmetro MaxIter (número de iterações) utilizado no procedimento GRASP sequencial para cada instância de dados são apresentados na Tabela 3.

Tabela 3 - Parâmetros GRASP Sequencial

Instância	N Itens	Max Iter
knapPI_1_100_1000_1	100	1024
knapPI_1_500_1000_1	500	1024
knapPI_1_1000_1000_1	1000	1024
knapPI_1_2000_1000_1	2000	1024

Os parâmetros MaxIter (número de iterações) e *threads* utilizados no procedimento GRASP paralelo para cada instância de dados são apresentados na Tabela 4. A variação na quantidade de *threads* e número de iterações do GRASP foram feitas para verificar o impacto no tempo de execução.

Tabela 4 - Parâmetros GRASP Paralelo CPU

Instância	N Itens	*Threads*	Max Iter
knapPI_1_100_1000_1	100	4	256
knapPI_1_100_1000_1	100	32	32
knapPI_1_100_1000_1	100	16	64
knapPI_1_100_1000_1	100	64	16
knapPI_1_500_1000_1	500	4	256

knapPI_1_500_1000_1	500	32	32
knapPI_1_500_1000_1	500	16	64
knapPI_1_500_1000_1	500	64	16
knapPI_1_1000_1000_1	1000	4	256
knapPI_1_1000_1000_1	1000	32	32
knapPI_1_1000_1000_1	1000	16	64
knapPI_1_1000_1000_1	1000	64	16
knapPI_1_2000_1000_1	2000	4	256
knapPI_1_2000_1000_1	2000	32	32
knapPI_1_2000_1000_1	2000	16	64
knapPI_1_2000_1000_1	2000	64	16

Os parâmetros MaxIter (número de iterações), *threads* e Blocos utilizados no procedimento GRASP paralelo em GPU para cada instância de dados são apresentados na Tabela 5.

Tabela 5 - Parâmetros GRASP Paralelo GPU

Instância	**N Itens**	**Blocos**	**Threads**	**Max Iter**
knapPI_1_100_1000_1	100	32	32	1
knapPI_1_100_1000_1	100	64	16	1
knapPI_1_100_1000_1	100	16	64	1
knapPI_1_100_1000_1	100	16	32	2
knapPI_1_100_1000_1	100	32	16	2
knapPI_1_100_1000_1	100	16	4	16
knapPI_1_500_1000_1	500	32	32	1
knapPI_1_500_1000_1	500	64	16	1
knapPI_1_500_1000_1	500	16	64	1
knapPI_1_500_1000_1	500	16	32	2
knapPI_1_500_1000_1	500	32	16	2
knapPI_1_500_1000_1	500	16	4	16
knapPI_1_1000_1000_1	1000	32	32	1
knapPI_1_1000_1000_1	1000	64	16	1
knapPI_1_1000_1000_1	1000	16	64	1
knapPI_1_1000_1000_1	1000	16	32	2
knapPI_1_1000_1000_1	1000	32	16	2
knapPI_1_1000_1000_1	1000	16	4	16
knapPI_1_2000_1000_1	2000	32	32	1
knapPI_1_2000_1000_1	2000	64	16	1
knapPI_1_2000_1000_1	2000	16	64	1
knapPI_1_2000_1000_1	2000	16	32	2
knapPI_1_2000_1000_1	2000	32	16	2
knapPI_1_2000_1000_1	2000	16	4	16

O parâmetro que define o tamanho da RCL para todos os testes é igual a 10. Este valor foi escolhido empiricamente, variando o valor para a mesma instância de dados e avaliando seu impacto na qualidade do resultado e no tempo de execução. Para valores altos, a geração da solução inicial fica muito aleatória, piorando a qualidade dos resultados e elevando o tempo de execução. Para valores bastante baixos o método fica com características gulosas na fase de gerar a solução inicial.

4.1. Resultados GRASP Sequencial

Nesta seção são apresentados os resultados computacionais dos testes realizados utilizando a metaheurística GRASP em sua abordagem sequencial. Este teste se faz necessário para se ter uma base de comparação entre os demais algoritmos paralelos, a fim de analisar as diferenças no tempo de execução e qualidade da solução. Ressalta-se que todos os testes foram feitos repetindo 100 vezes cada instâncias de teste, no mesmo computador. Os resultados obtidos encontram-se reportados na Tabela 6. A coluna desvio percentual (**DP**) mede o quão distante cada resultado ficou do ótimo conhecido.

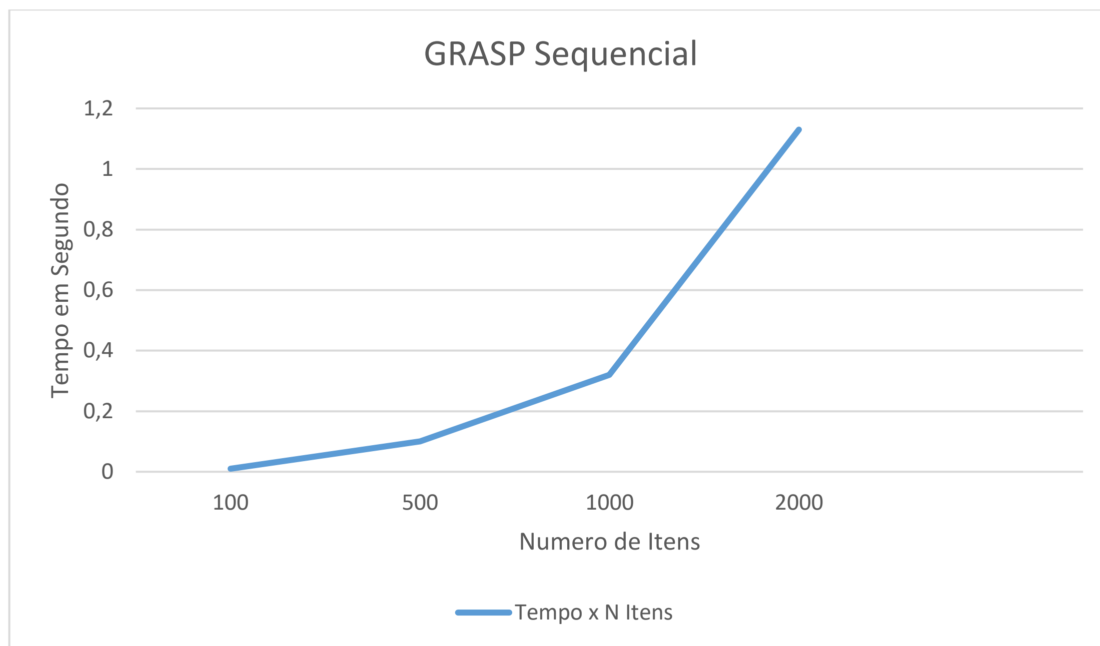
Tabela 6 - Resultados GRASP Sequencial

Instância	N Itens	MaxIter	Tempo Gasto (s)	Resultado Obtidos	Ótimo	DP
knapPI_1_100_1000_1	100	1024	0.01	9147	9147	0%
knapPI_1_500_1000_1	500	1024	0.10	28857	28857	0%
knapPI_1_1000_1000_1	1000	1024	0.32	54503	54503	0%
knapPI_1_2000_1000_1	2000	1024	1.13	110625	110625	0%

Segundo a Tabela 6, ao final dos testes, a abordagem utilizada para o desenvolvimento do GRASP sequencial apresentou um excelente desempenho em vista que, para instâncias de até 2000 itens o algoritmo encontrou o resultado que retorna o valor ótimo para todos os 100 testes em curto tempo de execução. Este tempo é resultado da média de todas as 100 execuções utilizando os mesmos parâmetros.

O Gráfico 1 mostra o crescimento do tempo de acordo com o crescimento da quantidade de itens que compõe o problema.

Gráfico 1 - Tempo x Itens GRASP Sequencial



4.2. Resultados GRASP paralelizado CPU

Nesta seção são apresentados os resultados computacionais (vide Tabela 7) obtidos ao fim dos testes executados utilizando o algoritmo GRASP em sua abordagem paralela rodando em CPU. A variação de número de iterações vezes a quantidade de *threads* será sempre igual a 1024 para manter o padrão de iterações nos testes. A coluna desvio percentual (**DP**) mede o quão distante cada resultado ficou do ótimo conhecido.

Tabela 7 - Resultados GRASP Paralelo CPU

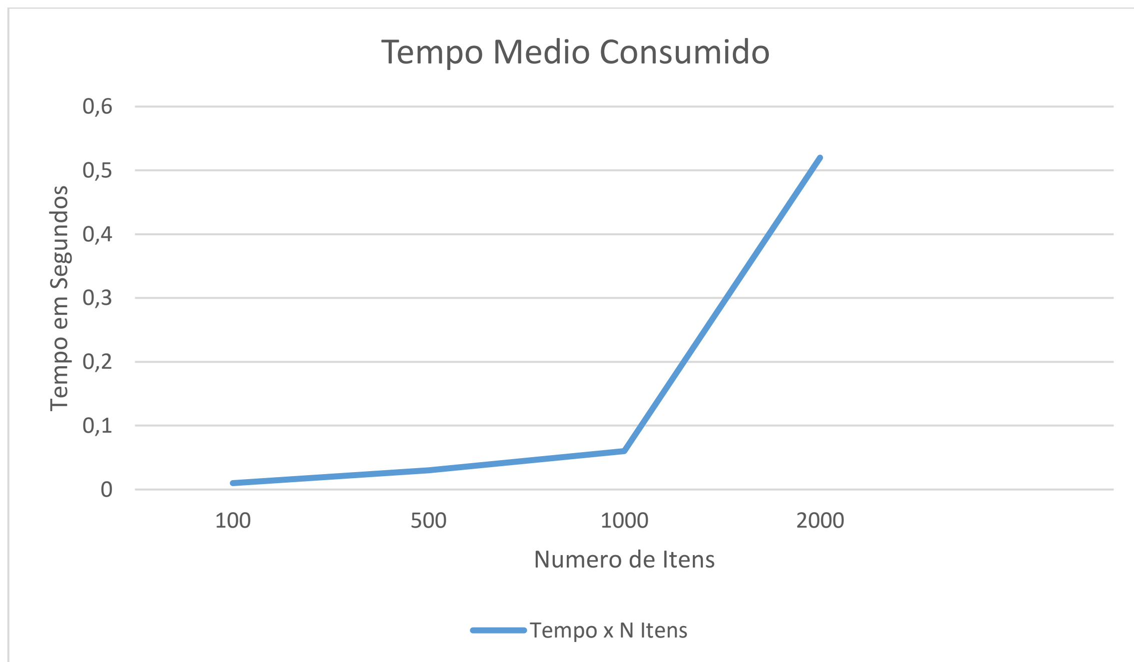
Instância	N Itens	MaxIter	*Threads*	Tempo Gasto (s)	Resultado	Ótimo	DP
knapPI_1_100_1000_1	100	256	4	0.01	8693	9147	4,96%
knapPI_1_100_1000_1	100	32	32	0.06	8940	9147	2,26%
knapPI_1_100_1000_1	100	64	16	0.03	8940	9147	2,26%
knapPI_1_100_1000_1	100	16	64	0.14	8940	9147	2,26%
knapPI_1_500_1000_1	500	256	4	0.03	28589	28857	0,92%
knapPI_1_500_1000_1	500	32	32	0.06	28794	28857	0,21%
knapPI_1_500_1000_1	500	64	16	0.04	28589	28857	0,92%
knapPI_1_500_1000_1	500	16	64	0.12	28794	28857	0,21%
knapPI_1_1000_1000_1	1000	256	4	0.06	54269	54503	0,42%
knapPI_1_1000_1000_1	1000	32	32	0.11	54405	54503	0,17%
knapPI_1_1000_1000_1	1000	64	16	0.08	54311	54503	0,35%
knapPI_1_1000_1000_1	1000	16	64	0.17	54405	54503	0,17%
knapPI_1_2000_1000_1	2000	256	4	0.17	110560	110625	0,005%
knapPI_1_2000_1000_1	2000	32	32	0.20	110527	110625	0,008%
knapPI_1_2000_1000_1	2000	64	16	0.18	110527	110625	0,008%
knapPI_1_2000_1000_1	2000	16	64	0.23	110500	110625	0,008%

Analisando os resultados após os testes serem executados utilizando o algoritmo GRASP paralelizado em CPU, a abordagem apresentou uma melhoria no tempo de execução para instâncias grandes. Em contrapartida, a qualidade dos resultados piorou não chegando no ótimo conhecido para nenhuma instância, ficando somente com uma solução boa perto da ótima conhecida.

Observando os resultados para a mesma instância alterando-se apenas a variável de *threads* e número e iterações tem-se que o melhor tempo de execução obtido se dá pela divisão do número de iterações por uma quantidade pequena de *threads*. Isso pode ser explicado pelo alto custo de se criar uma *thread* e inicializar a mesma e ainda no final do processo de todas as *threads* recuperar a melhor solução de todas elas.

O Gráfico 2 mostra o crescimento do tempo de acordo com o crescimento da quantidade de itens que compõe o problema.

Gráfico 2 - Tempo x Itens GRASP Paralelo CPU



4.3. Resultados GRASP Paralelo GPU

Nesta seção são apresentados os resultados encontrados ao final dos testes executados utilizando o algoritmo GRASP paralelo rodando em GPU utilizando a plataforma CUDA. Vários testes foram executados variando o número de iterações do GRASP, a quantidade de *threads* e blocos, sendo que o resultado da expressão *MaxIter * thread * blocos* seja sempre igual a 1024, buscando manter o padrão de geração de

resultados igual entre todos as abordagens. A coluna desvio percentual (**DP**) mede o quão distante cada resultado ficou do ótimo conhecido.

A Tabela 8 mostra os resultados obtidos para a instância **KnaPI_1_100_1000_1** com 100 itens executando o GRASP em GPU.

Tabela 8 - Resultados GRASP Paralelo GPU 100 Itens

Instância	Itens	MaxIter	*Threads*	Blocos	Tempo (s)	Resultado	Ótimo	DP
knapPI_1_100_1000_1	100	1	32	32	0.12	8940	9147	2,26%
knapPI_1_100_1000_1	100	1	16	64	0.11	8940	9147	2,26%
knapPI_1_100_1000_1	100	1	64	16	0.11	8940	9147	2,26%
knapPI_1_100_1000_1	100	2	32	16	0.11	8940	9147	2,26%
knapPI_1_100_1000_1	100	2	16	32	0.11	9147	9147	0%
knapPI_1_100_1000_1	100	16	4	16	0.13	9147	9147	0%

O algoritmo GRASP paralelo sendo executado na GPU configurado com 16 iterações, 4 *threads* e 16 blocos, e também com 2 iterações, 16 *threads* e 32 blocos conseguiu encontrar o valor ótimo conhecido para a instancia com 100 itens.

A Tabela 9 mostra os resultados obtidos para a instância **KnaPI_1_500_1000_1** com 500 itens executando o GRASP em GPU.

Tabela 9 - Resultados GRASP Paralelo GPU 500 Itens

Instância	Itens	MaxIter	*Threads*	Blocos	Tempo (s)	Resultado	Ótimo	DP
knapPI_1_500_1000_1	500	1	32	32	0.16	28805	28857	0,18%
knapPI_1_500_1000_1	500	1	16	64	0.16	28754	28857	0,35%
knapPI_1_500_1000_1	500	1	64	16	0.15	28702	28857	0,53%
knapPI_1_500_1000_1	500	2	32	16	0.18	28794	28857	0,21%
knapPI_1_500_1000_1	500	2	16	32	0.16	28691	28857	0,57%
knapPI_1_500_1000_1	500	16	4	16	0.34	28805	28857	0,18%

Já para a instancia de 500 itens o GRASP ficou atrás 0.18% da solução ótima conhecida com um tempo de execução igual a 0.16 segundos.

A Tabela 10 mostra os resultados obtidos para a instância **KnaPI_1_1000_1000_1** com 1000 itens executando o GRASP em GPU.

Tabela 10 - Resultados GRASP Paralelo GPU 1000 Itens

Instância	Itens	MaxIter	*Threads*	Blocos	Tempo (s)	Resultado	Ótimo	DP
knapPI_1_1000_1000_1	1000	1	32	32	0.25	54481	54503	0,04%
knapPI_1_1000_1000_1	1000	1	16	64	0.24	54481	54503	0,04%
knapPI_1_1000_1000_1	1000	1	64	16	0.24	54481	54503	0,04%
knapPI_1_1000_1000_1	1000	2	32	16	0.26	54481	54503	0,04%
knapPI_1_1000_1000_1	1000	2	16	32	0.23	54481	54503	0,04%
knapPI_1_1000_1000_1	1000	16	4	16	0.60	54405	54503	0,017%

Para 1000 itens, o GRASP encontrou uma solução 0.04% pior que o ótimo conhecido, com um tempo de execução igual a 0.48 segundos em sua melhor configuração sendo 32 *threads*, 32 blocos e 1 iteração, e também para 64 *threads*, 16 blocos e 1 iteração.

A Tabela 11 mostra os resultados para a instância **KnaPI_1_2000_1000_1** com 2000 itens executando o GRASP em GPU.

Tabela 11 - Resultados GRASP Paralelo GPU 2000 Itens

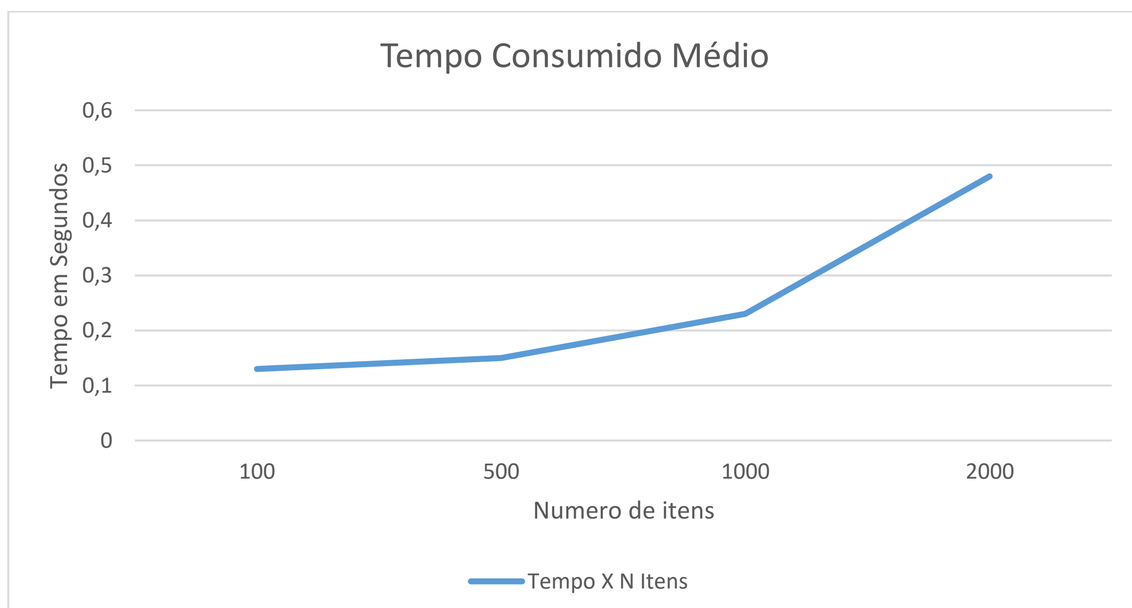
Instância	Itens	MaxIter	*Threads*	Blocos	Tempo (s)	Resultado	Ótimo	DP
knapPI_1_2000_1000_1	2000	1	32	32	0.48	110579	110625	0,04%
knapPI_1_2000_1000_1	2000	1	16	64	0.51	110577	110625	0,04%
knapPI_1_2000_1000_1	2000	1	64	16	0.48	110577	110625	0,04%
knapPI_1_2000_1000_1	2000	2	32	16	0.66	110570	110625	0,04%
knapPI_1_2000_1000_1	2000	2	16	32	0.52	110570	110625	0,04%
knapPI_1_2000_1000_1	2000	16	4	16	1.91	110570	110625	0,04%

Para a instância contendo 2000 itens, o GRASP paralelo em GPU obteve também um bom resultado, chegando bem próximo do ótimo conhecido diferenciando somente 0.04% da solução ótima da instancia com um tempo de execução igual a 0.48 segundo.

Os resultados obtidos pela execução do algoritmo GRASP paralelo em GPU apresentou um melhor tempo de execução se comparado com a abordagem sequencial. Se comparado com o algoritmo paralelo sendo executado na CPU, o tempo de execução gasto pela GPU é maior. Isso ocorre devido ao tempo de execução que se leva para alocar um espaço de memória na GPU para cada vetor como o vetor solução e vetor de itens, copiar todos os dados para a GPU e ao final do fluxo de execução no *device*, copiar de sua memória global os resultados obtidos para a memória RAM.

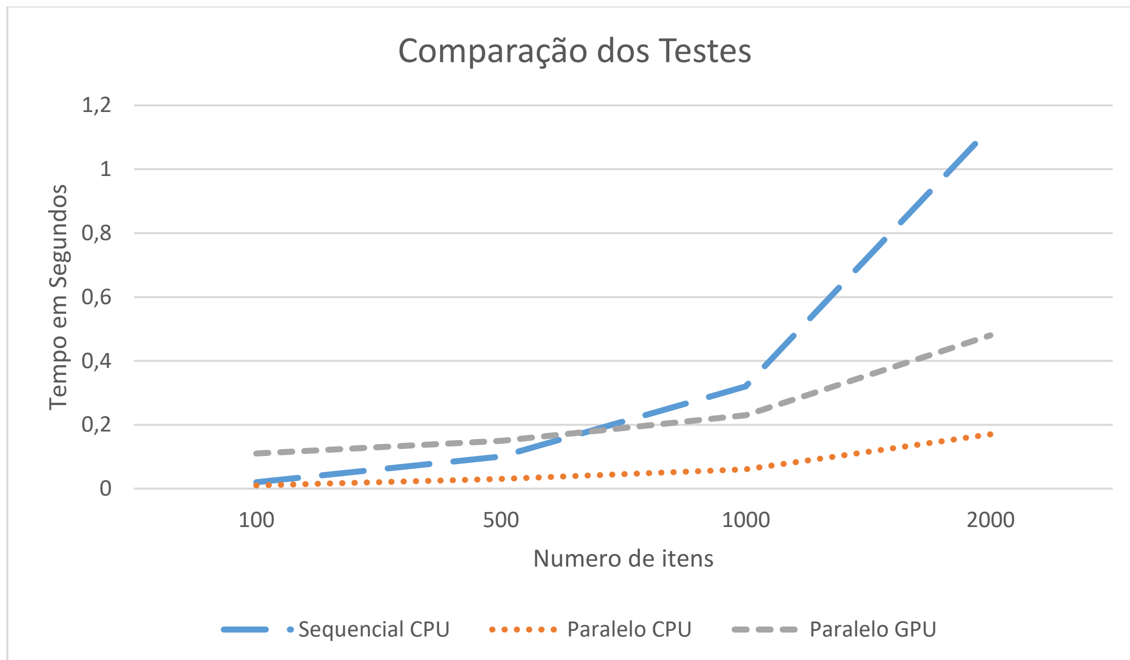
O Gráfico 3 mostra o crescimento do tempo de acordo com o crescimento das quantidades de itens.

Gráfico 3 - Tempo x Itens GRASP Paralelo GPU



Analisando a variação de *threads* por bloco e o número de iterações do GRASP segundo os testes executados no *device*, é notável que quanto maior o número de iterações e menor o número de *thread* por bloco maior é o tempo de execução. Essa variação causa um menor paralelismo no fluxo de execução na GPU, evidenciando a latência de acesso na memória global uma vez que, cada *thread* em seu bloco necessita fazer mais leitura e escrita. O Gráfico 4 mostra a comparação do crescimento do tempo segundo a quantidade de itens das três abordagens sequencial na CPU, paralelo na CPU e GPU.

Gráfico 4 - Comparação dos Testes



4.4. Limitações de Hardware

Seguindo o algoritmo proposto, é alocado na memória constante do *device* um vetor com a estrutura de dados criada e chamada de item. Cada posição do vetor armazena o valor do tipo inteiro, peso do tipo inteiro e o ganho do tipo *float*. Na linguagem C, o tipo inteiro e *float* ocupam um espaço de 4 bytes cada um. Logo cada espaço do vetor ocupa 12 bytes. Colocando 5.000 itens no vetor e alocando na memória constante o mesmo ocuparia 5.000 x 12 = 60.000 bytes ou 60kb, quase todo o espaço disponível para armazenamento que é igual a 64 kb (vide figura 12).

Cada *thread* em seu tempo de execução aloca em sua memória local disponível, que é dependente da configuração segundo a quantidade de *threads* e blocos e suas respectivas dimensões totalizando 512kb no total, o vetor RCL, um vetor do tipo de dados inteiro e com tamanho passado por parâmetro e escolhido empiricamente sendo de tamanho 10. Também é alocado um vetor responsável pelo método Roleta. Tal vetor

armazena em cada posição um dado do tipo inteiro e outro do tipo *float*, tendo o tamanho do vetor igual ao número de itens que pertence a solução. Cada *thread* então, aloca em sua memória local restrita 40 bytes (vetor RCL) + k * 8 bytes sendo k igual ao número de itens que compõe a solução.

5. Conclusão

Segundo o conteúdo desta monografia, é possível concluir que o percurso trilhado neste trabalho teve como primeiro momento o estudo do problema proposto, um problema clássico de otimização combinatória conhecido com Problema da Mochila juntamente com algumas abordagens de solução utilizando programação paralela. Tendo conhecido o problema, partiu-se para o estudo da metaheurística GRASP, um algoritmo multipartida que permite a paralelização proposto para a resolução do problema. De posse destes conhecimentos, o passo seguinte foi estudar programação paralela usando a biblioteca **Pthread** e a plataforma CUDA com ênfase nos aspectos de *hardware* e *software* visando sua melhor utilização. Por fim, após a realização de diversos experimentos, os resultados foram expostos, analisados e comentados.

Como resultado prático obtido neste trabalho tem-se a otimização do tempo de execução do algoritmo GRASP em relação a sua versão sequencial, a qual necessitou 1.13 segundos para resolver o problema proposto para a instância de 2000 itens. O melhor tempo de execução obtido para a versão paralela em CPU foi de 0.17 segundos e na versão em CUDA 0.48 segundos.

No que diz respeito às limitações deste trabalho, é importante elencar o fato de que o objeto de estudo se limitou ao problema da mochila binária com 1 dimensão. As outras variantes do problema como múltiplas mochilas ou mochilas multidimensionais certamente aumentariam a complexidade do problema, especialmente pelo fato de limitações de memória da GPU utilizada, implicando em um estudo mais aprofundado da plataforma CUDA e do *hardware* da GPU NVIDIA afim de buscar soluções para tais limitações ficando como trabalhos futuros.

Como conclusão deste trabalho, o mesmo cumpriu os objetivos de estudar, implementar e avaliar a plataforma CUDA para a resolução do problema da mochila utilizando a metaheurística GRASP com uma abordagem sequencial e paralela. Os resultados obtidos foram satisfatórios, o algoritmo conseguiu resolver o problema em pouco tempo de execução e encontrar o valor ótimo para quase todas instâncias testadas. Porém, a diferença no tempo de execução entre o algoritmo utilizando a plataforma CUDA e o utilizando a abordagem paralelo em CPU foi até certo ponto surpreendente. Esperava-se que o algoritmo em CUDA tivesse um tempo de execução melhor.

O alto custo computacional de mudar o fluxo de execução da CPU para a GPU devido a necessidade de alocar e instanciar os dados para processamento no *device*, a latência de acesso a memória global, copiar os resultados gerados no *device* para a memória RAM e fazer a busca do melhor resultado explicam o tempo maior de execução se comparado com a abordagem paralelo na CPU.

Fica como trabalhos futuros o estudo mais aprofundado da plataforma CUDA e novas abordagens como estratégia de execução na GPU, lidando com a hierarquia de memória e fluxo de execução buscando diminuir o tempo de execução. Também o desenvolvimento de novos algoritmos em CUDA para resolver problemas clássicos da computação como *Simulated Annealing* para resolução de alocação de recursos, multiplicação de produto interno de matrizes com grandes massas de dados entre outros.

Referências

ALVARENGA. Fabiano Vieira; ROCHA, Marcelo Lisboa. Melhorando o Desempenho da Metaheurística GRASP Utilizando a Técnica Path-Relinking: Uma Aplicação para o Problema da Árvore Geradora de Custo Mínimo com Grupamentos. Goiânia, 2006. XXXVIII Simpósio Brasileiro Pesquisa Operacional.

ALVES, Pedro Geraldo M. R.; Implementação eficiente em GPUs da avaliação homomórfica de programas em máquinas de instrução única. Instituto de Computação. Universidade Estadual de Campinas. Campinas - SP, Brasil. Setembro de 2014.

ANDRADE, Lisieux Marie Marinho dos Santos. Novas abordagens sequencial e paralela da metaheuristica C-GRASP aplicadas à otimização global contínua. 2013. 87p Dissertação de Pós-graduação em Informática. Centro de Informática. Universidade Federal da Paraíba, 2013.

CALDAS, Ruiter Braga; ZIVIANI, Nivio. Projeto e Analise de Algoritmos. Departamento de Ciencia da Computação. Universidade Federal de Minas Gerais. Belo Horizonte. 2004.

CARVALHO, Erica da Costa Reis; SILVA, Michelli Marlane. Otimização do Problema da Mochila. 2016. 14p. Trabalho de Conclusão de Curso. Departamento de Ciencia da Computação. Universidade Presidente Antônio Carlos, 2016.

Corporation Nvidia. CUDA Zone. Developer Nvidia. Nvidia. Disponível em: <https://developer.nvidia.com/cuda-zone>. Acessado em Junho de 2018.

DANTAS, Bianca de Almeida. Metaheurísticas para o Problema da Mochila Multidimensional. 2016. 110p. Tese de Doutorado. Faculdade de Computação, Universidade Federal de Mato Grosso do Sul, 2016.

DAVID B. Kirk, WEN-Me W. Hwu; Programando para processadores paralelos – Uma abordagem prática a programação de GPU. Rio de Janeiro: Elsevier, 2011.

FEOFILOFF, Paulo. O problema da mochila booleana. Instituto de Matemática e Estatística da USP, 13 de abril de 2015. Disponível em: <https://www.ime.usp.br/~pf/analise_de_algoritmos/aulas/mochila-bool.html >. Acessado em Junho de 2018.

Nvidia. CUDA – Programação paralela facilitada. Disponível em: <http://www.nvidia.com.br/object/cuda_home_new_br.html>. Acessado em Junho de 2018.

P. ROCHA, Kassiane de Almeida; F FILHIO. Luciano José Varejão. INTRODUÇÃO AO CUDA UTILIZANDO MÉTODOS NUMÉRICOS. 2010. 125p. Trabalho de Conclusão de Curso. Faculdade de Ciencia da Computação. Centro Universitário Vila Velha, 2010.

PISINGER, Davide; Instances of 0/1 Knapsack Problem. 2005. Disponível em: <http://artemisa.unicauca.edu.co/~johnyortega/instances_01_KP/ >. Acessado em Junho de 2018.

RAIDL, G. R., “A unified view on hybrid metaheuristics”. In F. Almeida et al., editores, Proceedings of the Hybrid Metaheuristics Workshop, volume 4030 of LNCS, páginas 1-12. Springer, 2006.

RESENDE, M; RIBEIRO, C. Greedy Randomized Adaptive Search Procedures. Handbook of metaheuristics, v. 57, p 219-249, 2003.

RESENDE, Mauricio Guilherme de Carvalho; SILVA, Ricardo Martins de Abreu. GRASP: Procedimentos de Busca Gulosos, Aleatórios e Adaptativos. In: Metaheurística em Pesquisa Operacional. Curitiba, PR: Omnipax, 2013. cap. 1, p. 1-20.

ROCHA, Marcelo Lisboa; APLICAÇÕES DE ALGORITMOS PARALELOS E HÍBRIDOS PARA O PROBLEMA DE ÁRVORE DE STEINER EUCLIDIANA NO R^n. 2008. 99p. Tese de Doutorado. Departamento de Engenharia Elétrica. Universidade Federal do Rio de Janeiro, 2008.