PONTIFÍCIA UNIVERSIDADE CATÓLICA DE MINAS GERAIS
Programa de Pós-graduação em Engenharia Elétrica

Vinícius Oliveira e Silva

Implementação Paralela Eficiente do Algoritmo Simplex Otimizado em GPU-CUDA

Belo Horizonte
2017

Vinícius Oliveira e Silva

Implementação Paralela Eficiente do Algoritmo Simplex Otimizado em GPU-CUDA

Dissertação apresentada ao Programa de Pós-graduação em Engenharia Elétrica da Pontifícia Universidade Católica de Minas Gerais, como requisito parcial para obtenção do título de Mestre em Engenharia Elétrica.

Orientador: Prof. Dr. Carlos Augusto Paiva da Silva Martins

Belo Horizonte
2017

FICHA CATALOGRÁFICA
Elaborada pela Biblioteca da Pontifícia Universidade Católica de Minas Gerais

Silva, Vinícius Oliveira e
S586i Implementação paralela eficiente do algoritmo simplex otimizado em GPU-CUDA / Vinícius Oliveira e Silva. Belo Horizonte, 2017.
45 f.: il.

Orientador: Carlos Augusto Paiva da Silva Martins
Dissertação (Mestrado) – Pontifícia Universidade Católica de Minas Gerais. Programa de Pós-Graduação em Engenharia Elétrica

1. Computação. 2. Programação linear. 3. Pesquisa operacional. 4. Algorítmos computacionais. 5. Programação (Computadores). 6. Programação paralela (Computação). 7. Computação de alto desempenho. I. Martins, Carlos Augusto Paiva da Silva. II. Pontifícia Universidade Católica de Minas Gerais. Programa de Pós-Graduação em Engenharia Elétrica. III. Título.

CDU: 681.3-11
Ficha catalográfica elaborada por Rosemary Socorro Hosken - CRB 6/3170

Vinícius Oliveira e Silva

Implementação Paralela Eficiente do Algoritmo Simplex Otimizado em GPU-CUDA

Dissertação apresentada ao Programa de Pós-graduação em Engenharia Elétricada Pontifícia Universidade Católica de Minas Gerais, como requisito parcial para obtenção do Título de Mestre em Engenharia Elétricado Programa de Pós-graduação em Engenharia Elétrica.

Prof. Dr. Carlos Augusto Paiva da Silva Martins (Orientador) - PUC Minas

---

Prof. Dr. Luís Fabrício Wanderley Góes- PUC Minas

Prof. Dr. Luiz Ramos - PUC Minas

Belo Horizonte, 30 de Novembro de 2017

AGRADECIMENTOS

Inicialmente, agradeço aos meus pais Ana e Ivo, meu irmão Gustavo, por terem acreditado e me encorajado na minha capacidade para conseguir concluir esta jornada mesmo após alguns obstáculos.

Ao meu orientador e amigo Prof. Carlos Augusto, por todos os ensinamentos, acadêmicos e sociais, e pela confiança depositada em mim durante o curso, serei eternamente grato.

Aos amigos que fiz no PPGEE, principalmente ao Fábio Cordeiro, Ana Martins, Harison Silva e André Monteiro por terem me ajudado e incentivado na conclusão do curso e por todas as horas de apoio dedicadas ao estudo em conjunto nas matérias concluídas durante do curso.

À PUC Minas, pelo ensino de qualidade e corpo docente tão competente disponibilizado para nós, alunos, que tivemos a sorte de conviver e aprender com a certeza de que estávamos experimentando de um dos melhores ensinos do país.

À todos meus amigos, da Cemig, Simply e àqueles que fazem parte da minha vida, meu muito obrigado.

"I haven't slept for two days
I've bathed in nothing but sweat
And I've made hallways
Scenes for things to regret
My friends they come
And the lines they go by
Tonight I'm gonna rest my chemistry"
- Interpol, Our Love to Admire, Rest My Chemistry - 2007

RESUMO

O processamento de propósito-geral em unidades de processamento gráfico têm sido largamente utilizado e atingindo melhorias de desempenho consideráveis em várias áreas do conhecimento, ao passo que continua a evoluir desde seus primeiros estágios. Este trabalho apresenta uma implementação modificada do método Simplex usando o poder computacional trazido pelas unidades de processamento gráfico *(GPU)* usando o framework da NVIDIA para programação em *GPU*, o "Compute Unified Device Architecture *(CUDA)*. Os resultados alcançados mostraram que a implementação em paralelo em *GPU* atingiram 15.6x e 22.3x de speedup máximo ao medirmos o tempo total (tempo de transferência de dados mais tempo de trabalho do *kernel*) e apenas o tempo da computação do *kernel* respectivamente, ao compararmos com a implementação tradicional sequencial utilizando *CPU*.

Palavras-chave: CUDA, GPU Computing, GPGPU, Linear Programming, Operational Research, Optimization, Parallel Computing, Simplex

ABSTRACT

The general-purpose graphics processing unit programming has been widely used and has achieved performance improvements across many knowledge domains as it keeps evolving since its earlier stages. This paper presents an implementation of a modified Simplex method using the computational power brought from graphics processing unit (GPU) computing using the NVIDIA framework for GPU programming, compute unified device architecture (CUDA). The results achieved shown that the parallel GPU implementation reached 15.6x and 22.3x maximum speedup measuring the overall time (data transfer plus kernel work time) and the kernel computation time respectively, in comparison with the standard sequential implementation using central processing unit (CPU).

Keywords: CUDA, GPU Computing, GPGPU, Linear Programming, Operational Research, Optimization, Parallel Computing, Simplex

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

Algoritmos de otimização são cada vez mais utilizados em várias áreas do conhecimento, como engenharia, finanças, dentre outras (NASH, 2000). A programação linear é um dos métodos matemáticos usados para encontrar soluções, viáveis ou ótimas, para problemas de otimização (DANTZIG, 1963). Em cenários reais de aplicação, onde o modelo matemático do problema pode conter centenas ou milhares de restrições, a complexidade e o tempo de execução desses algoritmos crescem de forma exponencial, levando a um custo computacional elevado e um tempo de execução não hábil para tais demandas. Com o intuito de se minimizar os custos computacionais de um problema de otimização denso, essas demandas são submetidas a algoritmos que trazem resultados não ótimos para o problema, porém num tempo de execução polinomial.

Os recentes estudos na área de computação heterogênea e computação de propósito geral em *GPU* (*general-purpose graphics processing unit - GPGPU*) demonstram que a utilização dessa tecnologia de paralelismo na solução de problemas de programação linear permite encontrar soluções ótimas em tempos de execução significativamente menores (LALAMI; BOYER; EL-BAZ, 2011) (BIELING; PESCHLOW; MARTINI, 2010) (HAMZIC; HUSEINOVIC; NOSOVIC, 2011).

Citado na lista dos dez algoritmos mais importantes do século 20 (SULLIVAN; DONGARRA, 2000), o método Simplex se destaca entre os algoritmos utilizados para resolver problemas de programação linear. Contudo, seu custo computacional cresce proporcionalmente ao tamanho do problema, podendo chegar a complexidade exponencial. Dado que seu processo de iteração para resolução de problemas utiliza, essencialmente, matrizes, o algoritmo Simplex apresenta características necessárias para que sua implementação em GPU. No entanto, devido à curva de aprendizagem para implementações em GPU e alta complexidade do algoritmo, propomos uma implementação mais simples e eficiente, reduzindo a quantidade de iterações do algoritmo tradicional, e uso mais eficiente de memória. Com intuito de demonstrar a iteração do algoritmo, assim como validar os resultados e permitir o reuso das implementações, este trabalho também implementa uma programa com interface gráfica para demonstração do algoritmo, e uma biblioteca que poderá ser reutilizada. Em conjunto, o programa e a biblioteca podem ser utilizados como mecanismo de ensino do sistemas de otimização, assim como no ensino de implementações de algoritmos utilizando GPU.

Em resumo, os problemas motivadores deste trabalho são:

- Carência de implementação open-source de uma biblioteca para programação linear. Apenas uma implementação foi encontrada, em (FORREST, 2014), porém não há no descritivo a presença de paralelismo utilizando GPU para tal biblioteca.

- Curva de aprendizagem de paradigmas para computação de propósito geral em GPU rasa.

- Tempo para domínio e implementação computacional de um método de resolução de problemas de programação linear alto.

- Comparação entre duas abordagens do método Simplex, implementando ambas de modo sequencial em CPU e de modo paralelo utilizando GPU.

1.1 Objetivos

O principal objetivo deste trabalho é a implementação de uma melhoria algorítmica no método Simplex, que seja capaz de superar o desempenho da implementação tradicional do algoritmo. Tanto a implementação tradicional quanto a modificada são implementadas de modo sequencial, utilizando *CPU*, e de modo paralelo, utilizando *GPU*, esperando-se que as versões com melhoria algorítmica superem o desempenho em todos os casos, seja sequencial ou paralelo.

Um objetivo secundário deste trabalho é a possibilidade de se utilizar esta implementação como uma biblioteca de otimização para problemas de programação linear. Sua função será de abstrair a complexidade de se utilizar os recursos de computação de propósito geral em *GPU*, utilizando a *application programming interface* (*API*) *CUDA* como suporte à programação de propósito geral em unidades de processamento gráfico (*GPGPU*). A biblioteca desenvolvida é capaz de encontrar a solução ótima, se houver, para problemas de otimização de modelos de programação linear densos e de larga escala, com centenas ou milhares de variáveis e restrições, no menor tempo possível. Tal biblioteca pode ser reaproveitada de forma genérica para outros sistemas, de áreas de engenharia e afins, que necessitem solucionar problemas de programação linear.

1.2 Justificativas e Escopo

Do ponto de vista científico, este trabalho apresenta uma implementação eficiente de um método de programação linear para resolver problemas de otimização, densos ou esparsos, e de larga escala, utilizando recursos de computação paralela proporcionados pelo uso de processamento de propósito geral em *GPU*.

Outro ponto de vista importante, é o fato deste trabalho gerar uma biblioteca, assim como ferramentas auxiliares que podem ser utilizadas para futuros estudos de caso e pesquisas na área de otimização. Levando em consideração a vasta gama de áreas do conhecimento que demandam de sistemas capazes de resolver problemas de otimização de larga escala em tempo hábil, estes poderão ser beneficiados pela utilização dessa biblioteca. Todo o código desta implementação está disponibilizado num repositório público de códigos-fonte (SILVA, 2015), podendo beneficiar outras pesquisas da área de *GPGPU*, e livre para contribuições e/ou evoluções da biblioteca.

Este trabalho possui o seguinte escopo:

- Análise e construção de um mecanismo eficiente de resolução para problemas de programação linear de forma paralela, utilizando o processamento de propósito geral em *GPU*.

- Construção o mesmo algoritmo de forma sequencial em *CPU* para propósitos de comparação com a versão paralela.

- Construção de uma biblioteca que utiliza a versão paralela do algoritmo em *GPU* para propósito de reuso e interoperabilidade.

- Implementação de uma biblioteca capaz de receber dados manualmente e/ou importar dados através de um arquivo de entrada do tipo *mathematical programming system* (*MPS*). O padrão *MPS* é um formato de arquivo textual utilizado para descrever problemas de programação linear.

- Propor melhorias no algoritmo e/ou analisar possíveis técnicas de otimização para um melhor cenário paralelo de implementação.

1.3 Contribuições Complementares

Como contribuições técnicas, foram desenvolvidas as seguintes implementações e ferramentas auxiliares, disponíveis em (SILVA, 2015) visando atingir os objetivos propostos:

- Implementação com interface gráfica do método Simplex modificado. O programa desenvolvido visa mostrar, usando uma interface intuitiva, o processo do algoritmo proposto. Foi utilizada a linguagem Visual Basic .Net para a construção do programa.
- Implementação sequencial e outra paralela, usando *CUDA*, de uma abordagem tradicional do método Simplex em C++ e *CUDA*. Implementação sequencial e outra paralela, usando CUDA, do método Simplex modificado em C++ e *CUDA*.
- Gerador de problemas de programação linear com parâmetros de quantidade de variáveis, restrições e esparsidade parametrizáveis.
- Transcritor de problemas de programação linear para o padrão *MPS*.

1.4 Estrutura do Trabalho

No capítulo 2 deste trabalho é apresentado, de forma simplificada, o método Simplex original e a sua versão com modificação algorítmica. Ainda no capítulo 2, também é apresentado um programa com interface gráfica que visa demonstrar os passos do algoritmo modificado, com objetivo didático. No capítulo 3 são apresentados os principais trabalhos relacionados à aplicação da computação paralela em *GPU* em métodos de resolução de problemas de programação linear. No capítulo 4 é apresentada a implementação paralela em *GPU-CUDA* do algoritmo modificado. No capítulo 5 é apresentada a estrutura da biblioteca criada para invocar a implementação do Simplex. No capítulo 6 apresentamos os resultados. No sétimo capítulo são apresentadas as conclusões do trabalho.

2 O MÉTODO SIMPLEX

Apresentado em 1947 pelo matemático norte-americano George Bernard Dantzig, e publicado no livro de sua autoria (DANTZIG, 1963), o método Simplex ainda é o método mais utilizado na atualidade para resolver problemas de programação linear em várias áreas do conhecimento onde é aplicado.

Um problema de programação linear consiste na maximização, ou minimização, de uma função objetiva linear contendo (n) variáveis, submetidas a um conjunto de (m) restrições. Para existir espaço para um ambiente de otimização neste sistema de equações lineares, obrigatoriamente a função objetivo deve estar submetida a um conjunto de no mínimo $(n + 1)$ restrições, sendo (n) o conjunto de variáveis não-básicas da função objetivo. Formalmente, um problema de programação linear possui a seguinte descrição matemática:

$$\begin{aligned} \max\ & x_0 = cx', \\ \text{t.q.:}\ & A'x' \leq b', \\ & x' \geq 0, \end{aligned} \tag{2.1}$$

sendo

$$c' = (c_1, c_2, ..., c_n) \in \Re^n, \tag{2.2}$$

$$A' = \begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m,1} & a_{m,2} & \cdots & a_{m,n} \end{pmatrix} \in \Re^{m \times n} \tag{2.3}$$

e

$$x' = (x_1, x_2, ..., x_n)^T, \tag{2.4}$$

Sendo (n) a quantidade variáveis, e (m) a quantidade de restrições no sistema linear (vide (LALAMI; BOYER; EL-BAZ, 2011)).

Inicialmente, o primeiro passo do método Simplex consiste em introduzir variáveis artificiais, ou variáveis de folga, nas restrições do problema, com o objetivo de transformar as inequações das restrições em igualdades (DANTZIG, 1963). O resultado dessa alteração da forma padrão do sistema linear resulta no que se chama de forma canônica do problema inicial (DANTZIG, 1963). Somente a partir da forma canônica o método Simplex pode ser aplicado. A partir dessa forma canônica do sistema, é possível construir uma tabela no formato de matriz, denominada *tableau* Simplex. O *tableau* é utilizado para uma execução manual do método simplex, sendo utilizado para um melhor domínio e entendimento do algoritmo.

Utilizando a forma canônica obtida, o método Simplex aplicará uma sequência de operações de articulação, separado em duas fases. A primeira fase determina se o problema possui solução, gerando uma nova forma canônica na qual o algoritmo de segunda fase poderá ser executado. Caso a primeira fase de articulação tenha sucesso, o algoritmo executa uma segunda fase de articulação, que tem o objetivo de determinar a solução ótima do sistema (DANTZIG, 1963).

As duas fases de articulação do algoritmo, consistem na determinação de um elemento pivô (ou elemento permissível) dentro do *tableau*, cuja linha e coluna serão submetidas a um algoritmo de troca, que irá gerar um novo *tableau*. O algoritmo terminará sua execução quando determinar que o problema não possui solução, ou quando o problema

tem solução ótima.

Ao analisarmos o problema de programação linear numa óptica geométrica, sua representação é feita através de um polítopo pertencente a $\Re^n$ (vide Figura 1), formado pela intersecção de vários semi-espaços, que no caso são gerados a partir das restrições cujo problema de programação linear descreve, de dimensão (n). O polítopo obtido descreve a região permissível do problema, ou seja, a região cuja solução do problema se encontra. Todo o espaço fora do polítopo caracteriza a região não permissível do problema.

O método Simplex tradicional é iniciado a partir da solução trivial do problema, caracterizando um dos vértices desse polítopo. No decorrer do algoritmo, os valores obtidos para as variáveis descrevem um novo vértice deste polítopo, "vizinho" do vértice anterior. O algoritmo se baseia no fato de que, se existe uma solução ótima e única para o problema, esta solução se encontra em um dos vértices desse polítopo.

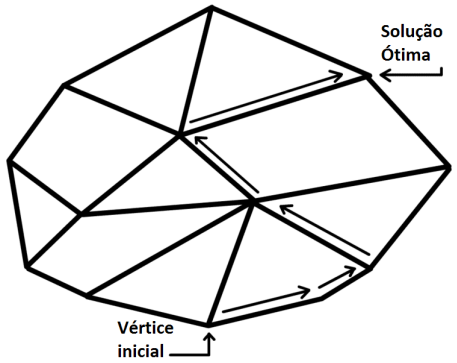

Figura 1 – Polítopo que descreve a região permissível. Fonte: (WIKIPEDIA, 2013)

2.1 Método Simplex - Alteração Algorítmica

A alteração algorítmica realizada no método Simplex tradicional consiste num algoritmo de duas fases, sendo ambas as fases do algoritmo submetidas a algoritmo de troca utilizado a fim de obter uma nova solução válida (PEDRYCZ PETR EKEL, 2011).

O algoritmo de primeira fase consiste na adaptação dos valores gerando uma nova solução parcial para o problema. Nessa etapa pode se chegar à conclusão que uma solução permissível não existe para o problema.

No algoritmo de segunda etapa pode se concluir que a região permissível descrita no problema é ilimitada, portanto, possuindo infinitas soluções.

Analogamente ao método tradicional do algoritmo Simplex, uma etapa de troca de variáveis no *tableau* também é realizada. Essa etapa caracteriza a obtenção de valores para as variáveis do problema, resultando num novo vértice adjacente ao anterior.

Para iniciar o algoritmo é necessário transformar a função objetivo num problema de minimização, caso necessário, e normalizar as restrições numa forma canônica, adicionando variáveis de folga e isolando os termos livres. Somente após essa etapa de normalização o *tableau* poderá ser construído.

A seguir, um exemplo de um problema de otimização, seguido pela sua forma norma-

lizada e a montagem do *tableau*.

$$\begin{aligned} \min Z &= 6x_1 + 12x_2 \\ 0.6x_1 + x_2 &\leq 600, \\ x_1 + x_2 &\geq 300, \\ x_2 &\geq 100, \\ x_i \geq 0 \text{ onde } i &= 1, ..., n \end{aligned} \tag{2.5}$$

Forma normalizada:

$$\begin{aligned} \text{F.O} \ &\rightarrow 0 - (-6x_1 - 12x_2) \\ x_3 &= 600 - (0.6x_1 + x_2), \\ x_4 &= -300 - (-x_1 + x_2), \\ x_5 &= -100 - (-x_2) \end{aligned} \tag{2.6}$$

A Fig.2 mostra o *tableau* para desenvolvimento do algoritmo.

Variáveis não básicas (VNB) / Variáveis Básicas (VB)	Membro Livre (ML)	x_1	x_2
$f(x)$	0	-6	-12
x_3	600	0,6	1
x_4	-300	-1	-1
x_5	-100	0	-1

Figura 2 – Tabela do Simplex modificado. Fonte: (PEDRYCZ PETR EKEL, 2011)

O algoritmo de primeira etapa (PEDRYCZ PETR EKEL, 2011):

1. Na tabela padronizada procuramos uma variável básica com membro livre negativo.
   (a) Se essa variável existe, então passamos para a fase 2 do presente algoritmo.
   (b) Se essa variável não existe, então passamos para a segunda etapa da solução do problema de programação linear.
2. Na linha que corresponde à variável com membro livre negativo, procuramos o elemento negativo da direita para esquerda.
   (a) Se o elemento negativo existe, então a coluna, onde está esse elemento, é escolhida como permissível.
   (b) Se o elemento negativo não existe, então a solução permissível não existe.
   (c) Se existem múltiplos elementos negativos, selecionamos o primeiro elemento.
3. Busca-se o elemento permissível a partir da identificação do menor quociente entre os membros livres que representam as variáveis básicas (VB).
4. Executar o algoritmo de troca.

O algoritmo de troca (PEDRYCZ PETR EKEL, 2011):

1. Calcula-se o inverso do elemento permissível (EP).
2. Multiplica-se toda a linha do EP pelo EP inverso.
3. Multiplica-se toda a linha do EP pelo valor negativo do EP inverso.
4. Marcar todas as sub-células superiores (SCS) da linha permissível e todas as sub-células inferiores (SCI) da coluna permissível.
5. Nas (SCI) vazias, multiplica-se a (SCS) marcada em sua respectiva coluna com a (SCI) marcada de sua respectiva linha.
6. Reescreva a tabela trocando de posição a variável não básica com a variável básica, ambas definidas como permissíveis na tabela anterior.
7. Todas as (SCI) da Linha e Coluna Permitida da tabela original deverão ser copiadas para suas respectivas (SCS) da nova tabela.
8. Somam-se as (SCI) com as (SCS) das demais células restantes da tabela original e seu resultado deverá ser copiado para sua respectiva (SCS) da nova tabela.
9. Retornar para o algoritmo de primeira etapa.

O algoritmo de segunda etapa (PEDRYCZ PETR EKEL, 2011):

1. Na linha F(x) procuramos um elemento positivo (não consideramos o membro livre).
   (a) Se o elemento positivo existe, então passamos para a operação 2 do presente algoritmo.
   (b) Se o elemento positivo não existe, então a solução ótima é obtida.
2. Na coluna permitida, correspondente ao elemento positivo escolhido, procuramos o elemento positivo fora da linha F(x).
   (a) Se o elemento positivo existe, então passamos para a operação 3 do presente algoritmo.
   (b) Se o elemento positivo não existe, então a solução ótima não existe, ou seja a solução é ilimitada.
3. Buscar o menor quociente por meio do resultado da busca de mínimo de relação do membro livre e seu respectivo elemento da coluna permitida. O termo livre e o elemento da coluna permissiva devem ter os mesmos sinais.
4. Executar o algoritmo de troca.

O algoritmo é executado de forma cíclica até que se encontre uma das seguintes situações terminais; o problema não tem solução, o problema tem infinitas soluções, ou o problema tem solução ótima.

2.2 Demonstração do Método Simplex com modificação algorítmica

Nesta seção demonstraremos a execução do algoritmo do método Simplex utilizando um programa desenvolvido durante este trabalho. Este programa visa demonstrar de forma gráfica cada passo do algoritmo descrito no tópico anterior, sendo uma forma didática de acompanhar o desenvolvimento das etapas do algoritmo.

Utilizaremos a seguinte função objetivo na demonstração do algoritmo:

$$\begin{aligned} \min\ Z = 6x_1 + 12x_2 \\ 0.6x_1 + x_2 \leq 600, \\ x_1 + x_2 \geq 300, \\ x_2 \geq 100, \\ x_i \geq 0 \text{ onde } i = 1, ..., n \end{aligned} \tag{2.7}$$

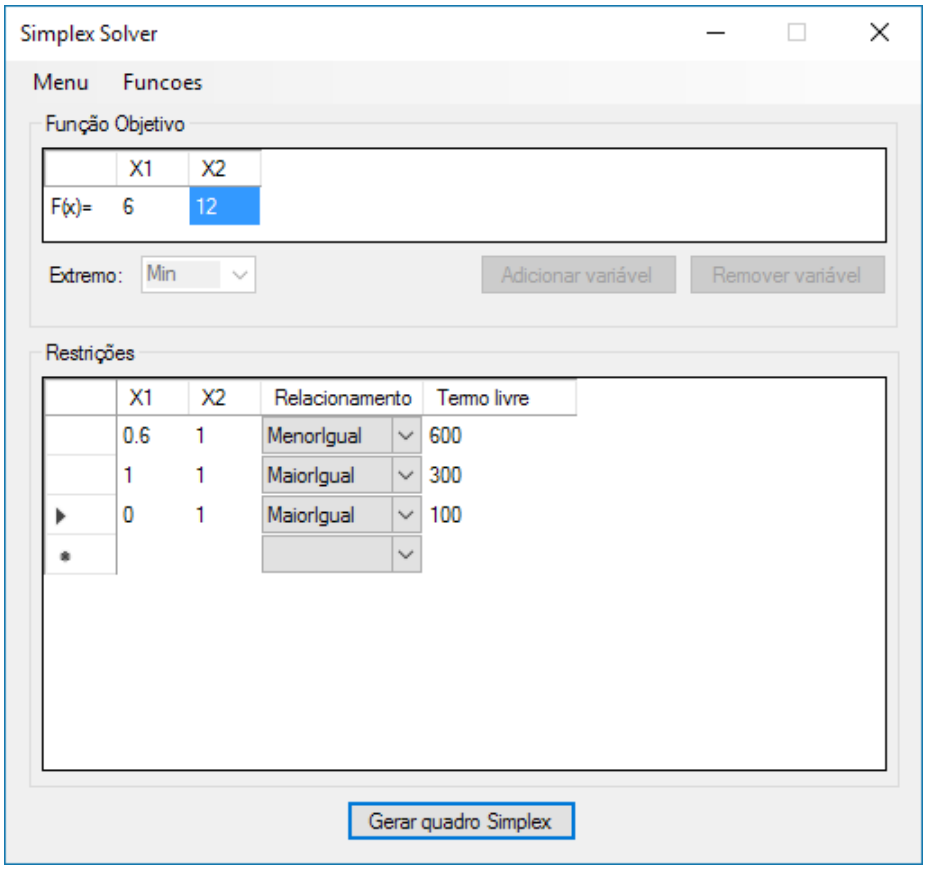

Figura 3 – Programa solucionador Simplex configurado com um problema de minimização. Fonte: Do próprio autor.

A Fig. 3 mostra a tela inicial do programa.

Nesta tela é possível adicionar variáveis à função objetivo, assim como adicionar restrições ao problema e os valores de suas respectivas variáveis e coeficientes. Para cada restrição adicionada, deverá ser informado o relacionamento da restrição com seu termo livre, possibilitando a configuração de restrições do tipo maior, menor, maior ou igual e menor ou igual.

Ainda é possível informar a direção da otimização, ou seja, se trata-se de um problema de minimização ou maximização da função objetivo. Não há um limite para a quantidade

de variáveis configuradas, sendo este restrito apenas à quantidade de memória disponível para a execução do problema.

Ao término da configuração do problema de otimização, devemos clicar no botão "Gerar Quadro Simplex"para iniciar o desenvolvimento do algoritmo.

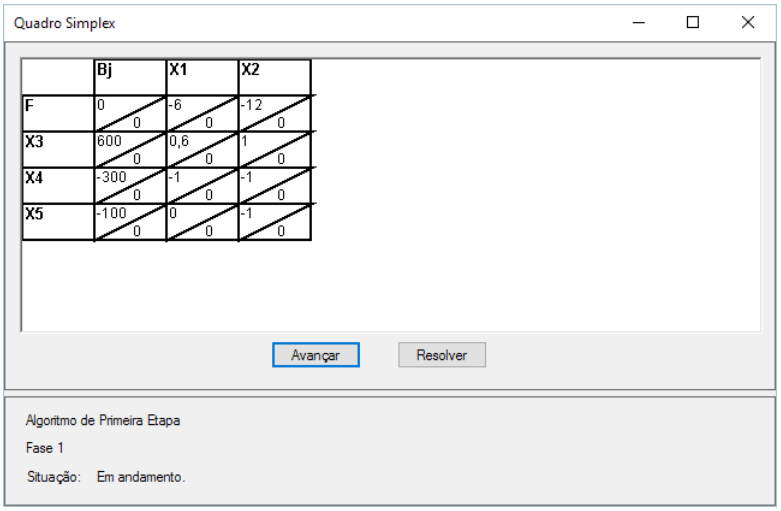

Figura 4 – Programa solucionador Simplex com a tela inicial do tableau Simplex. Fonte: Do próprio autor.

Ao clicar em “Gerar Quadro Simplex” a seguinte tela, mostrada na Fig. 4, será mostrada. Nesta tela é possível observar o quadro Simplex, já com a função objetivo devidamente normalizada na forma canônica, com os valores da função objetivo e restrições configuradas no programa.

Ainda nesta tela, o programa qual a etapa do algoritmo está em execução, assim como a fase corrente. É possível passar por cada fase do algoritmo clicando no botão “Avançar”, e dessa forma, acompanhar as decisões tomadas durante cada fase. Também pode-se resolver o problema de imediato clicando no botão “Resolver”.

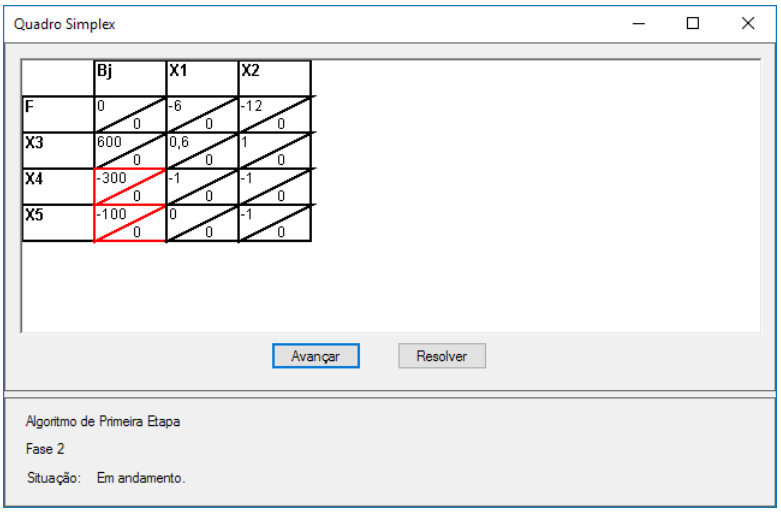

Figura 5 – Programa solucionador Simplex, após execução da fase 1 do algoritmo de primeira etapa. Fonte: Do próprio autor.

A Fig. 5 mostra o tableau após a execução da primeira fase do algoritmo de primeira etapa. Esta fase consiste na varredura da coluna dos termos livres em busca de valores negativos. O programa encontra dois valores aptos, na segunda e terceira linhas do tableau.

Neste caso, como existem dois valores aptos, utilizamos o primeiro valor encontrado, no caso, o valor (-300).

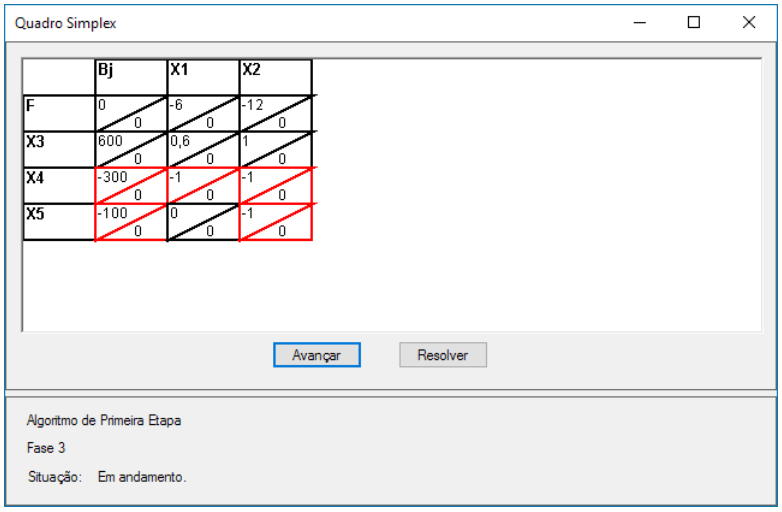

Figura 6 – Programa solucionador Simplex, após execução da fase 2 do algoritmo de primeira etapa. Fonte: Do próprio autor.

A Fig. 6 mostra o tableau após a execução da segunda fase do algoritmo de primeira etapa. Nesta etapa, o programa encontrou todas as variáveis básicas com coeficientes negativos na mesma linhas cujo termos livres também eram negativos. Na segunda linha do tableau temos dois valores negativos, ambos com o valor (-1). Neste caso, definimos a coluna permissível aquela com o menor relação entre o termo livre e o elemento da coluna candidata, cujos valores possuem o mesmo sinal. Como ambos os valores candidatos são (-1), escolhemos o primeiro valor da esquerda para a direita. Este valor será o elemento permissível. Passamos então para o algoritmo de troca.

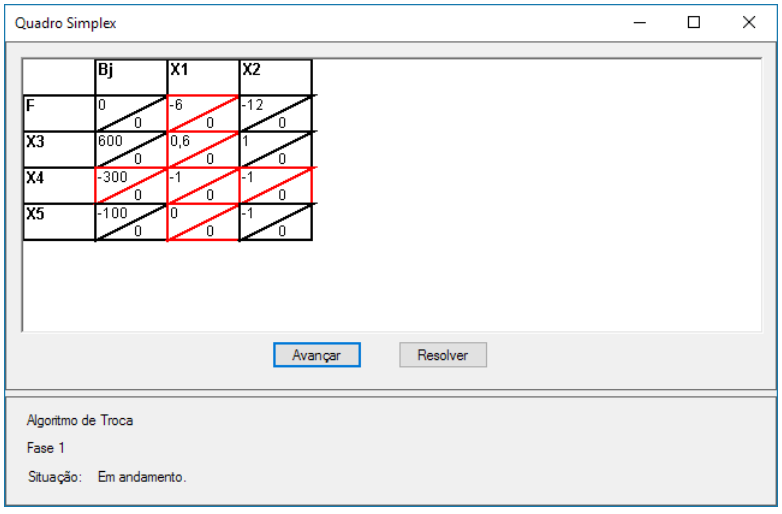

Figura 7 – Programa solucionador Simplex, executando a primeira etapa do algoritmo de troca. Fonte: Do próprio autor.

A Fig. 7 nos mostra a primeira fase do algoritmo de troca, destacando a linha e coluna do elemento permissível, elemento este localizado na célula $a_{3,2}$. Nesta etapa é calculado o inverso do elemento permissível.

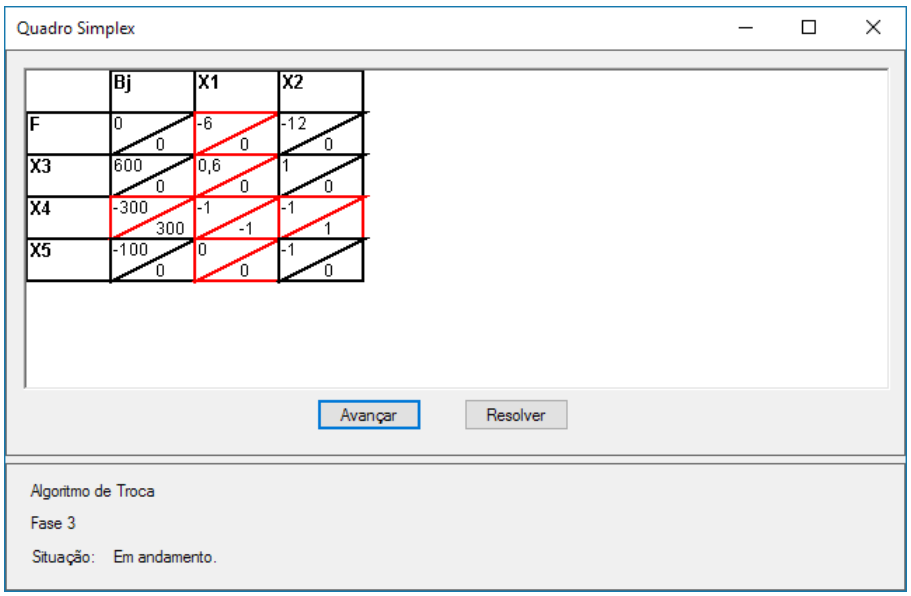

Figura 8 – Programa solucionador Simplex, executando a segunda etapa do algoritmo de troca. Fonte: Do próprio autor.

A Fig. 8 mostra a segunda etapa do algoritmo de troca sendo executado. Nesta etapa, a linha correspondente do elemento permissível é multiplicada pelo valor inverso do elemento permissível, atribuindo o valor na sub-célula inferior correspondente.

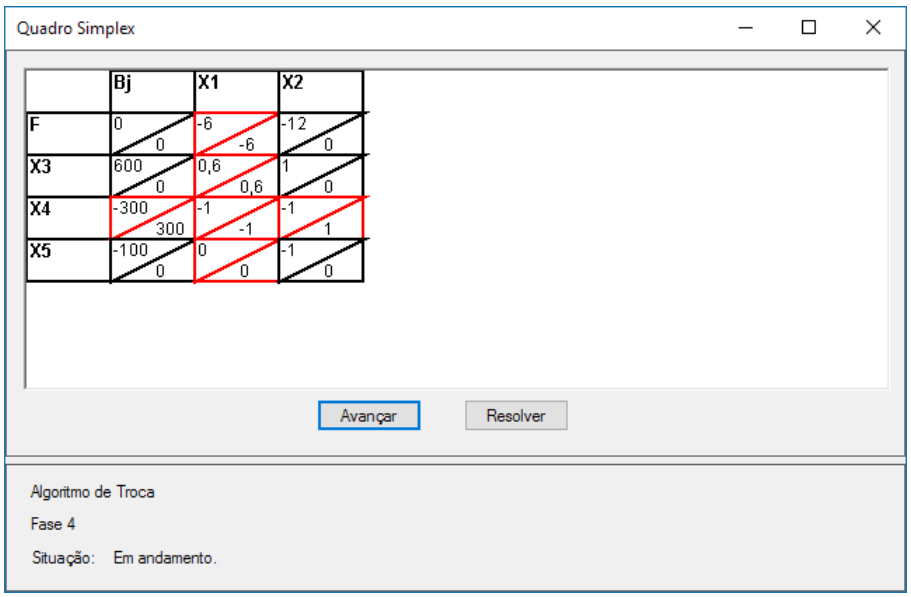

Figura 9 – Programa solucionador Simplex, executando a terceira etapa do algoritmo de troca. Fonte: Do próprio autor.

A Fig. 9 mostra a terceira etapa do algoritmo de troca, sendo que esta etapa realiza a multiplicação pelo valor negativo do inverso do elemento permissível por toda sua respectiva coluna, atribuindo o valor na sub-célula inferior correspondente.

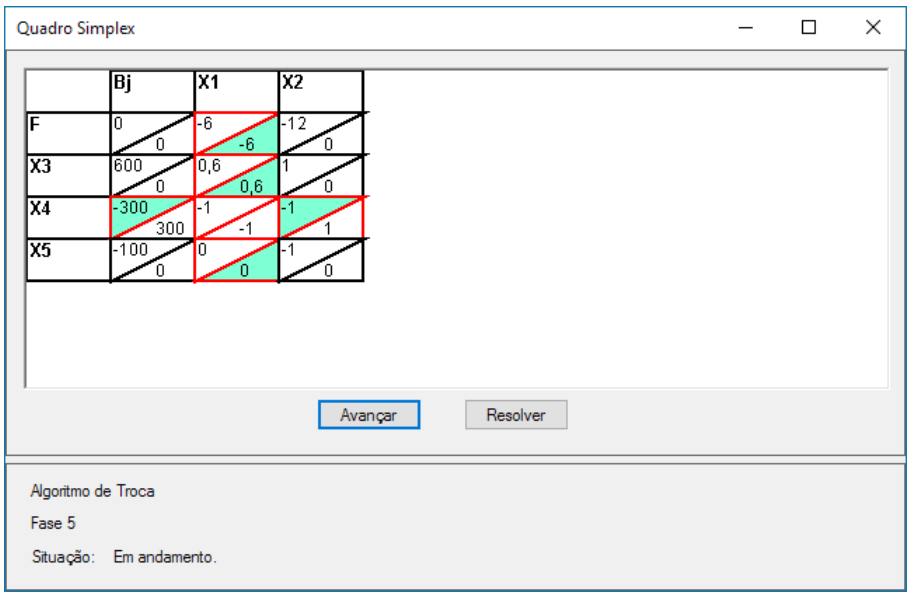

Figura 10 – Programa solucionador Simplex, executando a quarta etapa do algoritmo de troca. Fonte: Do próprio autor.

A quarta etapa do algoritmo de troca, mostrada na Fig. 10, é meramente ilustrativa, tratando apenas de marcar as sub-células superiores da linha permissível, assim como marcando as sub-células inferiores da coluna permissível. O programa destaca no tableau essas marcações.

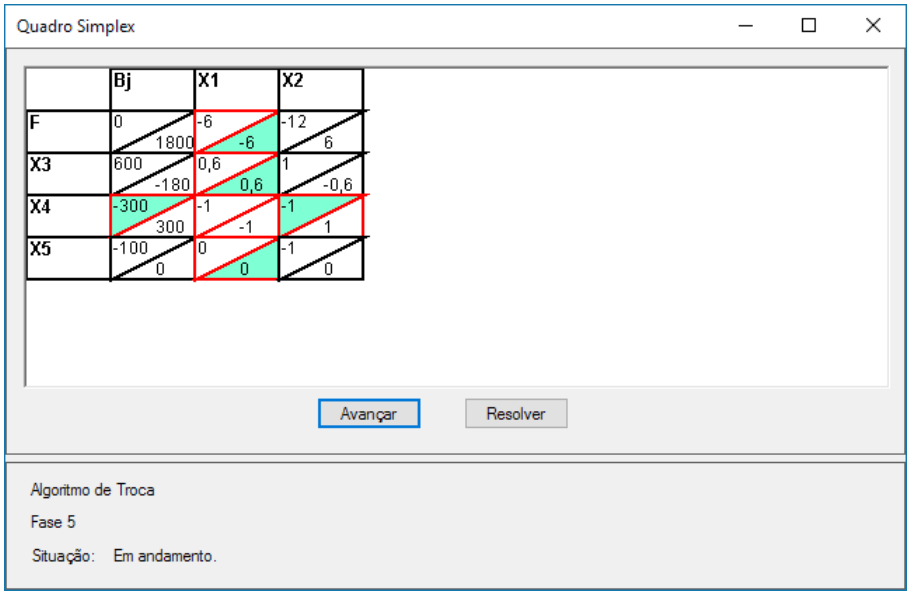

Figura 11 – Programa solucionador Simplex, executando a quinta etapa do algoritmo de troca. Fonte: Do próprio autor.

A quinta etapa do algoritmo de troca, mostrada na Fig. 11, é responsável por preencher os valores das demais sub-células inferiores vazias, realizando a multiplicação entre os valores das células marcadas de suas respectivas linha e coluna permissíveis.

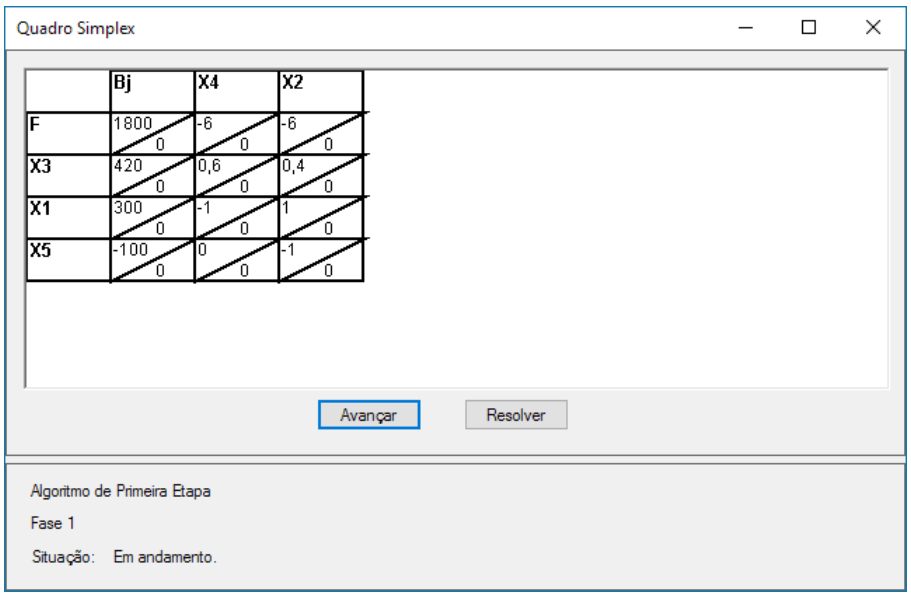

Figura 12 – Programa solucionador Simplex, executando a sexta etapa do algoritmo de troca. Fonte: Do próprio autor.

A última etapa do algoritmo de troca, mostrado na Fig. 12, revela o novo tableau obtido. Todas sub-células inferiores da linha e coluna permissíveis são copiadas para suas respectivas sub-células superiores, enquanto as demais células têm seu valor obtido através da soma entre suas sub-células inferiores com os valores da sub-célula superior, finalizando este ciclo do método Simplex.

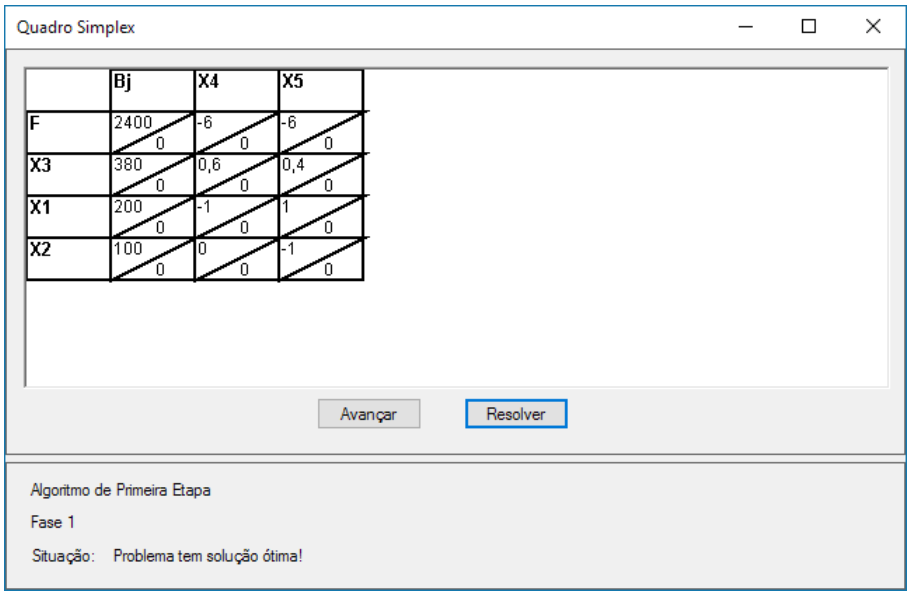

Figura 13 – Programa solucionador Simplex, exibindo a solução ótima encontrada para o problema de otimização. Fonte: Do próprio autor.

Após o reinício do algoritmo, a partir da primeira etapa e uma nova execução do algoritmo de troca, a solução ótima é obtida. O programa exibe então a mensagem de que o problema de otimização chegou ao seu fim, exibindo a mensagem com o veredito final do algoritmo. A Fig. 13 nos mostra a tela final da evolução do algoritmo.ss

3 TRABALHOS RELACIONADOS

3.1 Sparse Matrix Solvers on the GPU: Conjugate Gradients and Multigrid

Todos os trabalhos relacionados ao uso de *GPU* em algoritmos de otimização foram realizados com o propósito de encontrar qual o algoritmo produz o melhor comportamento e o melhor desempenho ao ser implementado de forma paralela. Um dos primeiros artigos encontrados durante a revisão bibliográfica, a tratar sobre melhorias de desempenho sobre manipulação de matrizes utilizando *GPU* foi em (BOLZ et al., 2005), implementando dois *kernels*, um solucionador baseado em gradientes-conjugados com matrizes esparsas e outro utilizando *multigrid*, visando beneficiar aplicações que utilizam estas operações em mecânica de fluídos e sólidos.

3.2 Implementing an interior point method for linear programs on a CPU-GPU system

O'Leary e Jung propuseram em (JUNG JIN HYUK; P., 2007), a resolução de problemas de programação linear utilizando o métodos de pontos interiores. Sua implementação consistia na utilização de computação heterogênea, fazendo uso tanto de *CPU* quanto da *GPU* por meio da *API CUDA*. Nesse caso, o poder computacional da *GPU* foi utilizado para operações feitas entre matrizes e fatoração de Cholesky. Diferentemente do método Simplex, que parte inicialmente da solução trivial para o sistema linear, o método de pontos interiores inicia sua busca a partir de pontos internos da região permissível. Em (JUNG JIN HYUK; P., 2007), ao comparar a versão implementada sequencialmente em *CPU* com a solução implementada usando *CPU-GPU*, foi conseguido um aumento sensível de desempenho para problemas de até 512 variáveis. No entanto, suas comparações de desempenho com problemas do repositório NETLIB (NETLIB, 1984) não registraram ganhos de desempenho, sugerindo que os problemas não eram grandes o suficiente para que um ganho de desempenho seja percebido.

3.3 Linear optimization on modern GPUs

Em (SPAMPINATO; ELSTER, 2009), Spampinato e Elster implementaram a versão revisada do algoritmo Simplex para problemas de programação linear em *GPU*. A implementação fez uso das bibliotecas NVIDIA CUBLAS (NVIDIA, 2008a) e NVIDIA LAPACK (NVIDIA, 2008b). Tais bibliotecas foram implementadas para acelerar a utilização de *GPU* em operações de álgebra linear na forma mais eficiente possível. Trabalhando com cargas de trabalho geradas de forma aleatória, com até 2000 variáveis e 2000 restrições, foi possível registrar *speedups* entre 2 e 2.5, ao se comparar com a versão sequencial em *CPU* do mesmo algoritmo.

3.4 An efficient GPU implementation of the revised Simplex method

Em (BIELING; PESCHLOW; MARTINI, 2010), os autores Bieling, Peschlow e Martini apresentaram uma nova implementação do algoritmo Simplex, em sua versão revisada. Esta nova implementação obteve um ganho máximo de 18x em relação ao solucionador

de problemas de programação linear *GNU Linear Programming Kit (GLPK)* (GNU..., 2000), em um ambiente de precisão *single-precision*.

3.5 Implementation and performance analysis of the Simplex algorithm adapted to run on commodity OpenCL enabled graphics processors

Ainda no campo da computação utilizando *GPU*, em (HAMZIC; HUSEINOVIC; NOSOVIC, 2011) foi utilizado a tecnologia *OpenCL* (KHRONOSGROUP, 2008), que se trata de uma plataforma de computação heterogênia, para se obter aproximadamente 28.9x de speed-up na implementação do Simplex Dual em comparação do Simplex tradicional em CPU para matrizes de até 8192 variáveis.

3.6 Efficient Implementation of the Simplex Method on a CPU-GPU System / Multi GPU Implementation of the Simplex Algorithm

Em (LALAMI; BOYER; EL-BAZ, 2011). Lalami, Boyer e El-Baz, realizaram uma implementação do algoritmo Simplex, em sua forma tradicional. Apesar de constatarem que o algoritmo simplex revisado geralmente possuir um desempenho melhor que a versão tradicional, para problemas densos e de larga escala seus desempenhos são semelhantes. O principal objetivo do trabalho foi implementar o algoritmo simplex num ambiente de precisão *double-precision*, e sem utilizar as bibliotecas da NVIDIA CUBLAS e LAPACK, no intuito de se obter desempenho superior. Utilizando uma carga de trabalho, gerada de forma aleatória, de problemas de programação linear densos, foi possível atingir um *speedup* de 12.5 numa placa de vídeo GTX 260 da NVIDIA. Estes mesmos autores, reutilizaram a implementação supracitada num ambiente com múltiplas GPU's, conforme demonstrado em (LALAMI; EL-BAZ; BOYER, 2011), conseguindo um ganho de até 24x de *speedup* usando duas placas de vídeo Tesla C2050. Concorrentemente, uma outra implementação, também com múltiplas *GPU*, do algoritmo tradicional foi citado em (MEYER; CHOPARD, 2011), porém, fazendo uso das bibliotecas CUBLAS para desempenho e apresentou resultados semelhantes.

3.7 A Branch-and-Bound algorithm using multiple GPU-based LP solvers

Em (MEYER; CHOPARD; ALBUQUERQUE, 2013), foi realizada uma implementação utilizando o método de *Branch-and-Bound* (B&B) para otimização de problemas lineares. O trabalho implementou uma abordagem híbrida, utilizando processamento em *CPU* e *GPU* durante o processamento do algoritmo, gerenciando a árvore do algoritmo (B&B) na *CPU* e a maior parte do método Simplex na *GPU* usando a biblioteca CUBLAS. Neste trabalho foi percebido que os ganhos usando a técnica apresentada somente eram percebidos em problemas suficientemente grandes e densos. Ao comparar o desempenho com o solucionador *Coin-or Linear Programming* (CLP) (FORREST, 2014), problemas esparsos obtinham melhor desempenho sobre a abordagem usando (B&B) e *GPU*.

3.8 Considerações Finais

Tendo em vista os trabalhos relacionados apresentados, podemos observar que a maior parte das implementações do método Simplex aplicadas de forma paralela ou híbrida, utilizando GPU, obtiveram resultados com desempenho superior. É perceptível que, nos primeiros trabalhos realizados nesta área os ganhos eram sensíveis ao passo que haviam diversas limitações de hardware da *GPU*, como compatibilidade em ambiente *double-precision*, e limite de memória, conforme demonstrados em (JUNG JIN HYUK; P., 2007) e em (SPAMPINATO; ELSTER, 2009). Com a rápida melhoria das arquitetura das da *GPUs* e a evolução das bibliotecas de computação de propósito geral em *GPU*, era esperado que o desempenho alcançado pelas implementações paralelas dos algoritmos fossem cada vez maiores.

4 IMPLEMENTAÇÃO DO SIMPLEX MODIFICADO EM *GPU-CUDA*

Nesta seção apresentam-se os detalhes da implementação paralela, utilizando *GPU-CUDA*, e os pontos de interesse que garantiram o bom desempenho do algoritmo.

Embasado nos trabalhos relacionados, e diante do método tradicional, foi identificado que a melhor forma de paralelizar o método seria através de uma implementação híbrida, utilizando tanto recursos de *CPU* quanto de *GPU*, sendo estes recursos aplicados nos pontos onde realmente demonstram melhor desempenho.

O método Simplex utilizado é semelhante em termos de separação em duas fases e determinação do elemento permissível, porém o mecanismo de pivoteamento (algoritmo de troca) e a utilização do *tableau* possuem mudanças em sua técnica. Em linhas gerais, o algoritmo utiliza duas operações básicas:

- Determinar o elemento permissível.
- Realizar o algoritmo de troca.

O *tableau* descrito no capítulo 2 utiliza uma matriz cujas células são subdividas. No mecanismo paralelo criado, foram utilizadas duas matrizes, uma para representar as subcélulas superiores (variável matrizSuperior) e outra para representar as subcélulas inferiores (variável matrizInferior).

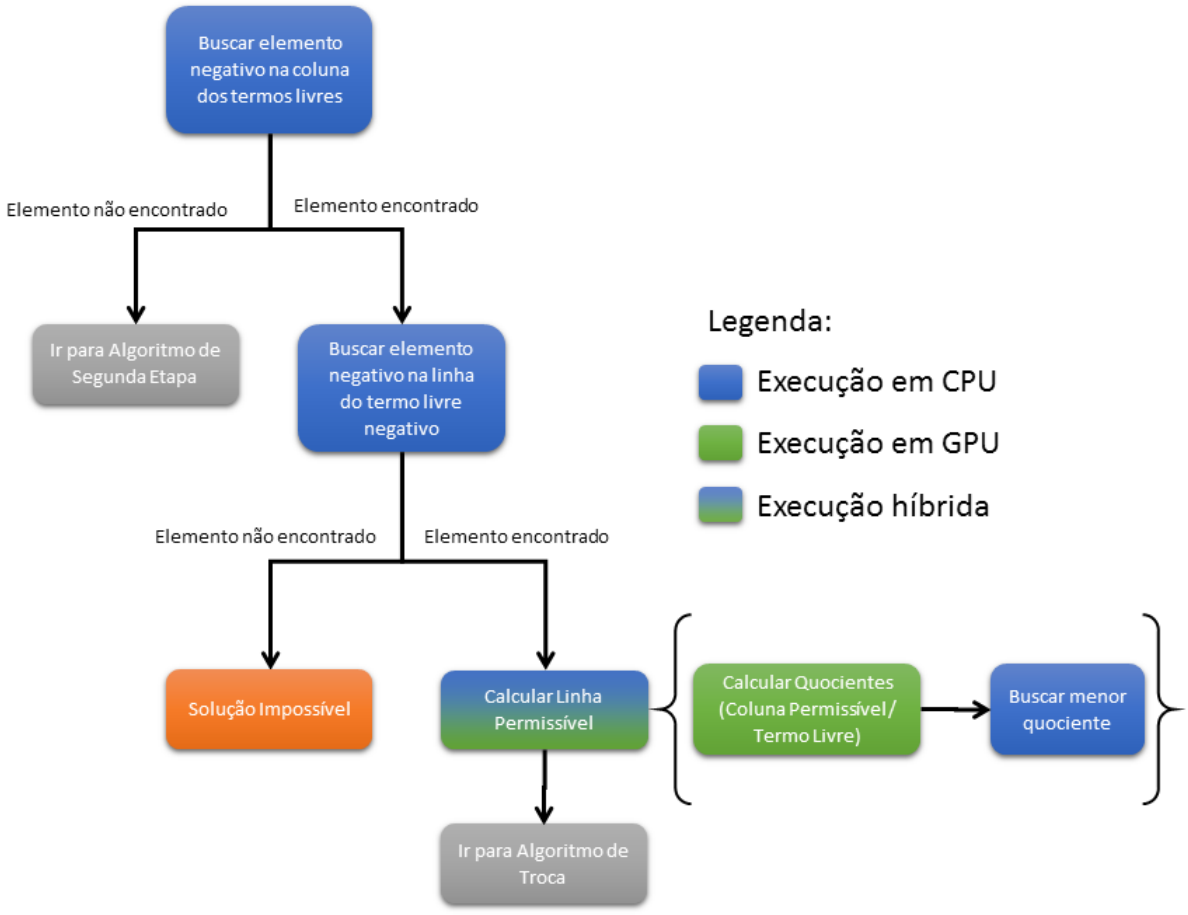

Figura 14 – Diagrama que descreve a execução do algoritmo de primeira etapa. Fonte: Do próprio autor.

Antes do início do algoritmo de primeira etapa, o sistema aloca e copia a matriz superior do *tableau* no dispositivo (*GPU*). É importante notar que, a matriz inferior do *tableau* só possui valores válidos durante o algoritmo de troca, quando são realizadas operações utilizando a coluna e linha permissíveis a fim de descobrir os novos valores da matriz superior. Dito isto, não é necessário alocar uma nova matriz para armazenar os valores das células inferiores, mas apenas clonar os valores dos vetores que descrevem os valores da linha e coluna permissíveis que são utilizados nos cálculos das demais células. Este é um fator fundamental no desempenho do algoritmo, tanto sequencial quanto paralelo utilizando *CUDA*, visto que uma matriz reduzida gera economia de memória e, portanto, uma redução na quantidade de dados a serem transferidos entre o *host* e *device*, e vice-versa. Além disso, a matriz reduzida acarreta na economia de tempo de trabalho realizado por cada *thread* do algoritmo paralelo em *GPU*.

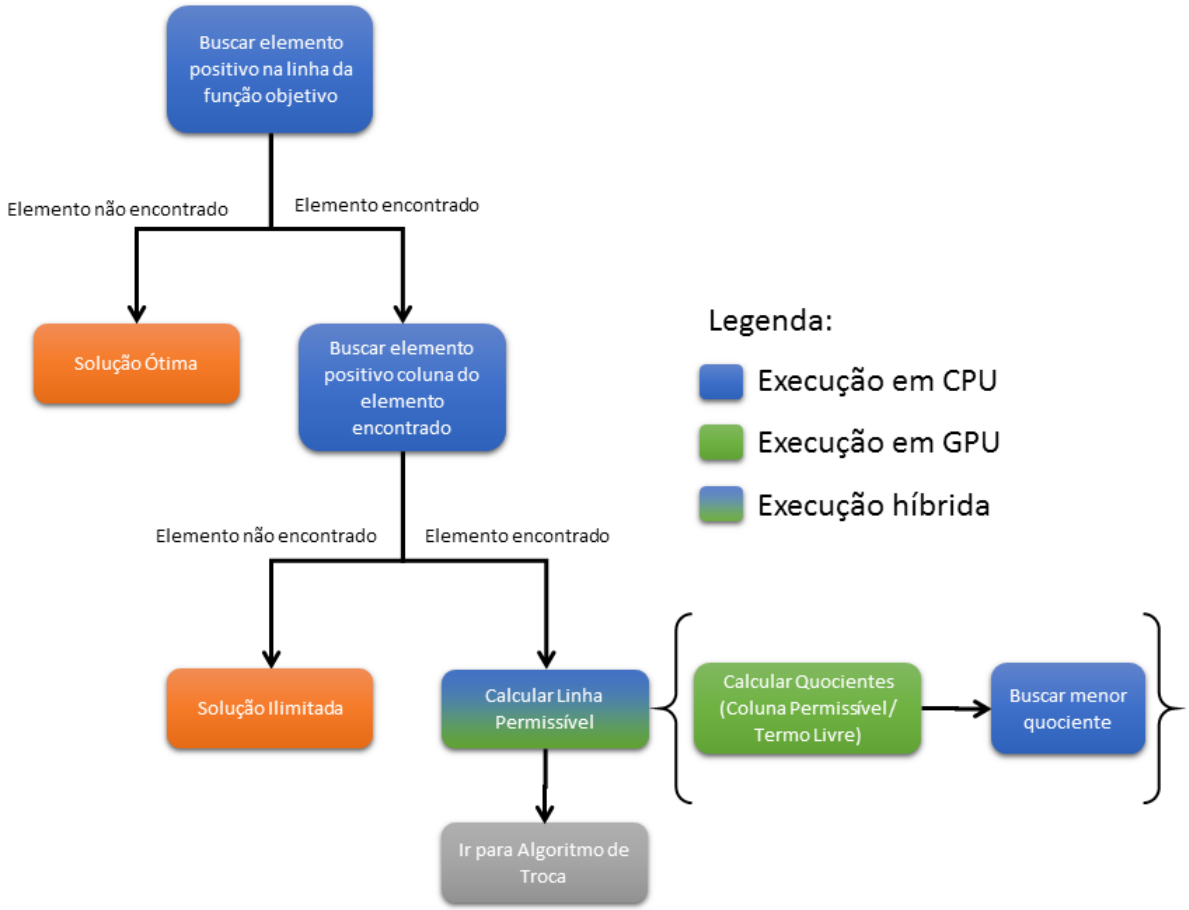

Figura 15 – Diagrama que descreve a execução do algoritmo de segunda etapa. Fonte: Do próprio autor.

Determinar o elemento permissível implica em descobrir sua linha e coluna permissíveis na matriz. Os algoritmos de primeira e segunda etapas descrevem como identificar esses índices permissíveis de maneira muito similar, sendo percorridas linha e coluna da matriz em busca de um determinado elemento de valor negativo e positivo respectivamente. A princípio, foi cogitada a utilização da técnica de *reduce* em *GPU* para otimizar essas etapas de busca de elementos. Porém, o algoritmo exige que se retorne o primeiro elemento negativo ou positivo da matriz, dependendo da etapa, ou seja, a técnica de *reduce* deveria ser aplicada de forma diferenciada para o correto funcionamento do algoritmo. Tomando a fase um do algoritmo de primeira etapa como exemplo, onde é necessário percorrer, no pior

caso, as m linhas da coluna de membros livres em busca de um elemento negativo, a técnica de *reduce* deveria ser aplicada sobre um vetor de índices de valores negativos da coluna de membros livres, para enfim determinar qual é o menor entre eles. Seria necessário inicialmente descobrir todos os índices cujos valores são negativos, o que acarretaria num custo computacional equivalente a uma busca direta utilizando *CPU*. Portanto, as fases de busca por elementos de específico valor foram implementadas em *CPU*, embasado nos resultados de (LALAMI; BOYER; EL-BAZ, 2011). As figuras Fig.14 e Fig.15 mostram as etapas dos algoritmos de primeira e segunda etapa, respectivamente, diferenciando a localização da execução das funções utilizando as cores azul, para execuções de código em CPU, e verde, para execuções de código em GPU.

Ainda na fase de determinação do elemento permissível, para determinar a linha permissível é necessário calcular o menor quociente entre a coluna permissível encontrada e os elementos da coluna de membros livres. Nesta fase foi possível aplicar o paralelismo em *GPU*, tendo em vista a aplicação da técnica de *map* para processar a divisão entre dois vetores de m elementos, a coluna permitida e a coluna de membros livres.

O seguinte pseudocódigo foi desenvolvido para retornar o vetor de quocientes:

Algorithm 1: Calcular Quocientes

```
   Entrada: vetorQuocientes, matrizSuperior, colunaPerm, totalColunas, totalLinhas
   /* obter índice da matriz relativo à thread atual                          */
1  i ← blockDim.x * blockIdx.x + threadIdx.x
   /* evitar operação em endereço inválido de memória                         */
2  if i == totalLinhas then
3  |  return
4  end
5  if matrizSuperior[i * totalColunas + colunaPerm] == 0 then
6  |  vetorQuocientes[i] ← −1;
7  else
8  |  vetorQuocientes[i] ←
   |    matrizSuperior[i * totalColunas]/matrizSuperior[i * totalColunas + colunaPerm];
9  end
```

Fonte: Do próprio autor.

O vetor de quocientes resultante dessa operação é copiado para o host, onde é feita uma busca pelo primeiro menor valor positivo de forma sequencial em *CPU*. Ao final deste processo, obtemos os índices que descrevem a localização do elemento permissível na matriz.

De posse do elemento permissível, o algoritmo inicia sua fase de atualização do *tableau*, utilizando o algoritmo de troca. Inicialmente, o algoritmo havia sido implementado realizando alocação das células superiores e inferiores, o que gerava uma matriz com o dobro de tamanho do problema original, o que é verificado também nas implementações tradicionais do método Simplex, levando em consideração que os vetores que descrevem as variáveis básicas também estão presentes no *tableau*. Porém, conforme dito anteriormente, os valores das células inferiores são apenas memória de cálculo para facilitar a abordagem didática deste algoritmo, portanto, foi realizada uma alteração no procedimento visando tanto o melhor desempenho do algoritmo, quanto uma economia de memória considerável. Inicialmente é necessário fazer a cópia dos vetores que descrevem os elementos da linha e coluna permissíveis.

O seguinte pseudocódigo descreve esse comportamento:

Algorithm 2: Copiar Linhas e Colunas Permissíveis

```
Entrada: matrizSuperior, vetLinhaPerm, vetColunaPerm
         linhaPerm, colunaPerm, totalColunas, totalLinhas
/* obter índice da matriz relativo à thread atual                    */
1  i ← blockDim.x * blockIdx.x + threadIdx.x
2  if i < totalColunas then
3  |  iLinhaPerm ← linhaPerm * totalColunas + (i%totalColunas)
4  |  vetLinhaPerm[i] ← matrizSuperior[iLinhaPerm]
5  end
6  if i < totalLinhas then
7  |  iColunaPerm ← i * totalColunas + colunaPerm
8  |  vetColunaPerm[i] ← matrizSuperior[iColunaPerm]
9  end
```

Fonte: Do próprio autor.

Após a cópia dos elementos da linha e coluna permissíveis, é possível concluir a fase do do algoritmo de troca, que se trata da atualização de todos os elementos da matriz. O seguinte pseudocódigo foi criado para computar os valores (vide o tópico descritivo do mecanismo de pivoteamento):

Algorithm 3: Algoritmo de Troca

```
Entrada: ep, matrizSuperior, vetLinhaPerm, vetColunaPerm
         linhaPerm, colunaPerm, totalColunas
/* obter índice da matriz relativo à thread atual                    */
1  i ← blockDim.x * blockIdx.x + threadIdx.x
2  iLinha ← i/totalColunas
3  iColuna ← i%totalColunas
/* se o índice pertencer a linha ou coluna permissíveis              */
4  if iLinha == linhaPerm ∨ iColuna == colunaPerm then
   |  /* se o índice for o elemento permissível                      */
5  |  if iLinha == linhaPerm ∧ iColuna == colunaPerm then
6  |  |  matrizSuperior[i] ← 1/ep
7  |  end
   |  /* se for elemento da linha permissível                       */
8  |  else if iLinha == linhaPerm ∧ iColuna ≠ colunaPerm then
9  |  |  matrizSuperior[i] ← matrizSuperior[i] * (1/ep)
10 |  end
   |  /* se for elemento da coluna permissível                       */
11 |  else if iLinha ≠ linhaPerm ∧ iColuna == colunaPerm then
12 |  |  matrizSuperior[i] ← matrizSuperior[i] * (−1/ep)
13 |  end
14 else
   |  /* calcular elemento da SCI da coluna permitida               */
15 |  eleColPerm ← vetColunaPerm[iLinha] * (−1/ep)
   |  /* calcular elemento da SCI atual                             */
16 |  eleSCI ← vetLinhaPerm[iColuna] * eleColPerm
   |  /* somar elemento da SCI inferior virtual com o elemento da posição atual */
17 |  matrizSuperior[i] ← matrizSuperior[i] + eleSCI
18 end
```

Fonte: Do próprio autor.

Ao final, a matriz superior que estava alocada na *GPU device* é copiada para a matriz superior no host, terminando a iteração. O algoritmo é então reiniciado a partir da primeira etapa. A Fig.16 mostra as etapas implementadas para a realização do algoritmo de troca.

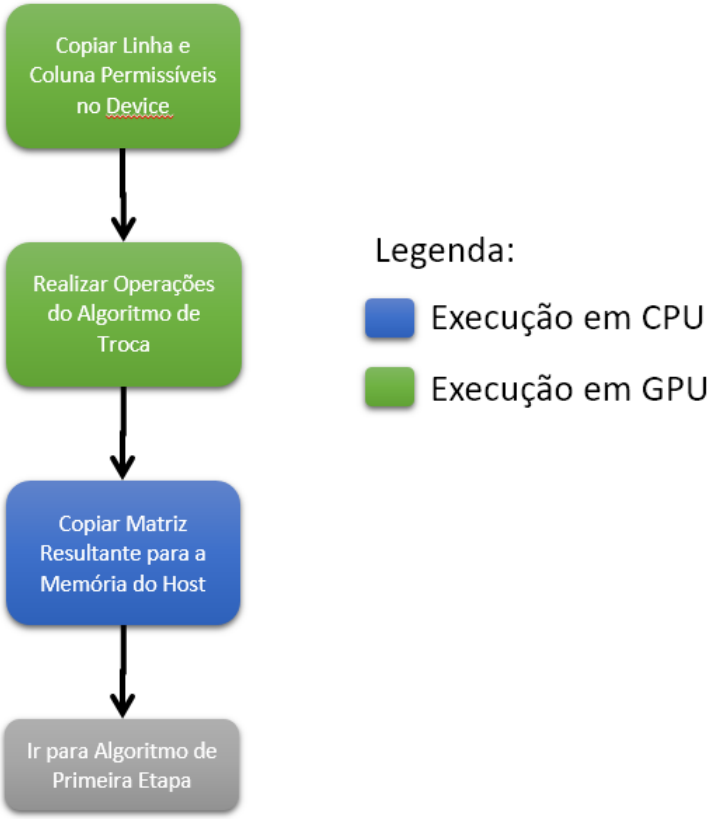

Figura 16 – Diagrama que descreve a execução do algoritmo de troca. Fonte: Do próprio autor.

5 BIBLIOTECA CUDASIMPLEX

Neste capítulo será apresentada as estruturas básicas da biblioteca desenvolvida durante o trabalho, visando dar uma perspectiva técnica e documentar as principais funções criadas.

5.1 Estrutura da Biblioteca

Um objetivo secundário deste trabalho é a criação de uma biblioteca para a resolução de problemas de programação linear, portanto, descreveremos brevemente, de forma técnica, a arquitetura e a utilização da mesma.

Para atingir esse objetivo foi necessário:

- Criação de estrutura de dados que represente de forma clara o problema de programação linear. Utilizando recursos de orientação a objeto do C++, foram criadas as seguintes classes:
  - ***Variavel:*** Classe responsável por armazenar o nome e o valor de coeficiente de uma variável.
  - ***Restricao:*** Classe responsável por representar uma restrição do problema de programação linear, armazenando um conjunto de variáveis indexados pelo respectivo nome, valor da constante do termo livre, sinal de desigualdade e uma variável básica que será utilizada para transformar a desigualdade da restrição em uma igualdade (forma canônica).
  - ***FObjetivo:*** Classe que representa a função objetivo do problema de programação linear, e encapsula todas as operações necessárias para descrevê-lo. Armazena todas as variáveis do problema, restrições, variáveis básicas criadas na fase de normalização, e um indicador de direção da otimização (maximizarminimizar).
- Fornecer funções para gerenciar a criação do problema de otimização. Foram criadas as seguintes funções:
  - ***addVariavel(Nome,Coeficiente):*** Função que adiciona uma variável na função objetivo.
  - ***addRestricao(Nome):*** Função que cria uma restrição no problema de otimização.
  - ***addVariavelRestricao(NomeRestricao, NomeVariavel, Coeficiente):*** Função que adiciona uma variável a uma restrição existente no problema.
  - ***setDesigualdadeRestricao(NomeRestricao, Desigualdade):*** Função que determina o valor de desigualdade de uma restrição, através das constantes *Maior*, *Menor*, *MaiorOuIgual*, *MenorOuIgual*, *Igual*.
  - ***setTermoLivreRestricao(NomeRestricao, TermoLivre):*** Função que seta o valor da constante (termo livre) de uma restrição existente no problema.
- Por fim, a criação de um controlador responsável por receber um problema de otimização e resolvê-lo. Para isso, foi criada uma classe chamada *SimplexSolver*, que recebe uma instância de *FObjetivo*. Ao invocar a função *SimplexSolver::otimizar()*, o

controlador irá realizar as operações de normalização do problema (forma canônica), e iniciará as etapas do método Simplex.

A Fig. 17 mostra o diagrama de classes dos modelos que representam uma função objetivo nos programas desenvolvidos.

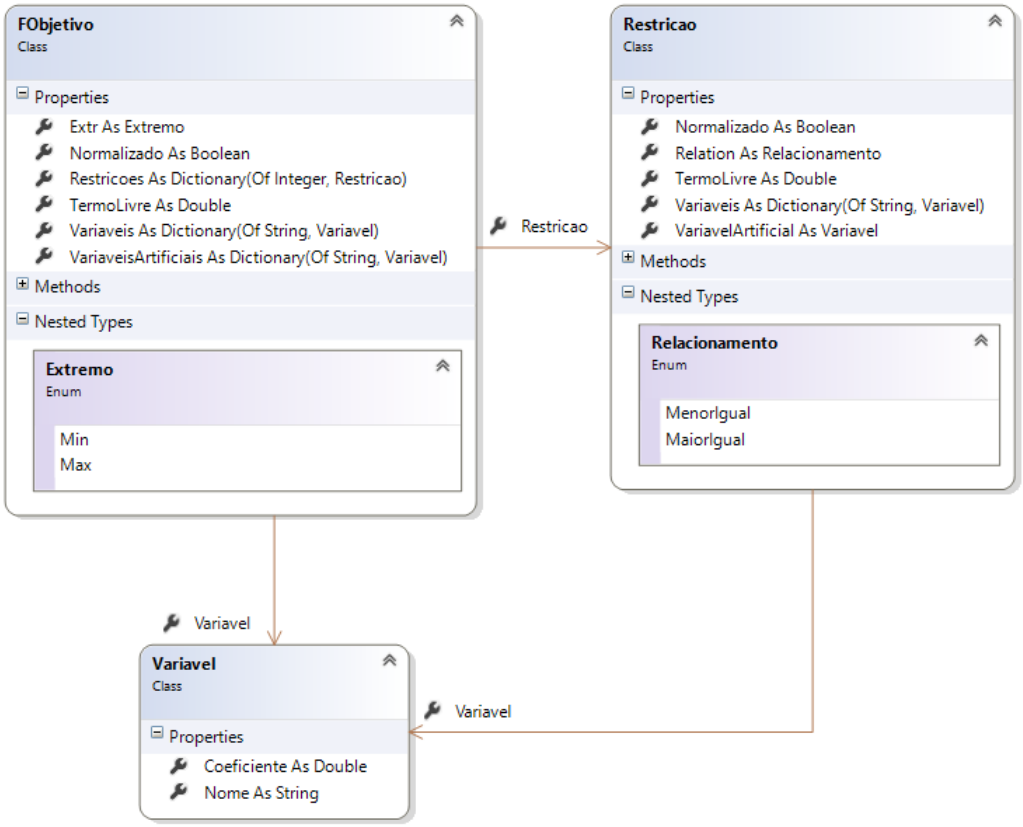

Figura 17 – Diagrama de classes que representam uma função objetivo. Fonte: Do próprio autor.

5.2 Utilização da Biblioteca

Utilizando o paradigma criado, é possível descrever a seguinte função objetivo,

$$\begin{aligned} \min\ Z = 6x_1 + 12x_2 \\ 0.6x_1 + x_2 \leq 600, \\ x_1 + x_2 \geq 300, \\ x_2 \geq 100, \\ x_i \geq 0 \text{ onde } i = 1, ..., n \end{aligned} \tag{5.1}$$

da seguinte forma:

```
FObjetivo funcao;
funcao.DirecaoOtimizacao = Minimizar;
funcao.addVariavel("x1", 6);
funcao.addVariavel("x2", 12);
```

A primeira restrição do problema pode ser configurada da seguinte forma:

```
funcao.addRestricao("Rest_1");
funcao.addVariavelRestricao("Rest_1", "x1", 0.6);
funcao.addVariavelRestricao("Rest_1", "x2", 1);
funcao.setDesigualdadeRestricao("Rest_1", MenorOuIgual);
funcao.setTermoLivreRestricao("Rest_1", 600);
```

Por fim, após a descrição do problema de programação linear, caberá apenas indicar ao controlador a função que será resolvida e invocar a função *SimplexSolver::otimizar()*.

```
SimplexSolver solver (&funcao);
solver.otimizar();
```

5.3 Programa Gerador de Problemas de Programação Linear

Para testar as implementações desenvolvidas, foram utilizados problemas gerados aleatoriamente. O gerador de problemas desenvolvido permite a parametrização quanto à quantidade de variáveis, restrições e a densidade da matriz a ser produzida. Os problemas gerados são escritos em arquivos no formato *Mathematical Programming System* (MPS - vide (Wilson; Rudin, 1992)). Esse formato permite a reutilização dos problemas gerados em outros *solvers* para atestar a assertividade dos resultados obtidos.

5.4 Considerações Finais

Neste capítulos mostramos as principais estruturas da biblioteca criada para resolução de problemas de programação linear. Também mostramos o quão simples é sua utilização, e portanto, conseguindo abstrair do programador a complexidade inerente do desenvolvimento de programas de alto desempenho que utilizem a biblioteca *CUDA* para programação em *GPU*. Dessa forma, o programador pode se concentrar na parte essencial do problema que deseja resolver, ignorando os aspectos mais complexos de um novo paradigma de programação em paralelo utilizando *CUDA*.

6 RESULTADOS

Neste capítulo serão apresentados os resultados obtidos com ambas as implementações do método Simplex, sua implementação tradicional, e outra com a modificação algorítmica.

6.1 Hardware Utilizado

Os testes foram realizados em duas máquinas, sendo que a primeira máquina contém a CPU 1, e a segunda máquina contém a CPU 2 e a *GPU* utilizada para os testes da implementação em *CUDA*.

Abaixo seguem as especificações técnicas dos recursos computacionais utilizados:

- CPU 1: AMD A8-3500M 1.50GHz (Quad Core)
- CPU 2: Intel Core i7-3615QM 2.30GHz (Octa Core)
- GPU: NVidia Geforce GT 650M 1024 MB GDDR5 - 900MHz - 384 *CUDA Cores*

6.2 Métrica Utilizada

Tendo em vista que o método Simplex soluciona os problemas de programação linear em uma quantidade não-determinística de iterações, não há relação direta entre o tamanho da matriz, que gera o *tableau*, e a quantidade de iterações utilizadas para resolver o problema. Caso a solução ótima do problema esteja próxima ao ponto de origem (ponto de partida do algoritmo), bastam poucas iterações para encontrar sua solução.

O grau de densidade da matriz está diretamente ligado ao tempo de resolução do problema, pois a quantidade de vértices a serem analisados numa matriz densa é superior ao de uma matriz de mesma dimensão, porém esparsa. Sendo assim, para medir com fidelidade o desempenho do algoritmo, em ambas as implementações, foi medido o tempo, em segundos, que um algoritmo demora para concluir uma iteração do algoritmo. Entende-se por "uma iteração", o tempo compreendido entre o início do algoritmo até a execução do algoritmo de troca, e assim sucessivamente.

O tempo de cópia dos dados entre a memória *host* e a memória do *device* está incluído no tempo total medido por iteração do algoritmo. É importante salientar que essa operação, de transferência de dados entre *host* e *device* ocorre a cada finalização do algoritmo de troca, atualizando a matriz do problema na memória do *host*, reiniciando os passos do algoritmo com a matriz atualizada.

6.3 Aspectos Metodológicos

Sobre as cargas de trabalho utilizadas, estas foram geradas utilizando o gerador de problemas de programação linear em format *MPS*. A quantidade de restrições e variáveis do problema são fixas, sendo sempre a mesma quantidade para gerarem matrizes quadradas de larga escala, portanto, ao falarmos que um problema tem dimensão 100, significa um problema de 100 variáveis submetidas à 100 restrições.

Quanto aos aspectos metodológicos da coleta de dados, cada problema foi submetido a uma bateria de três testes consecutivos, e ao final de cada bateria a média aritmética de

tempo por iteração do problema era calculada. É importante salientar que este trabalho não se preocupou em realizar ajustes finos nas execuções do algoritmo em *GPU-CUDA*, tais como melhorias de alocação memória compartilhada e/ou ajustes nas configurações do tamanho do bloco de *threads*. Tais ajustes podem, e devem trazer ganhos ainda maiores no que diz respeito ao desempenho do algoritmo.

6.4 Resultados: AMD A8-3500M 1.50GHz (Quad-Core) vs GPU

A seguir seguem os gráficos obtidos durante as medições de desempenho, seguidos das suas respectivas análises, utilizando a *CPU 1* (AMD):

A Fig. 18 demonstra o desempenho de uma implementação tradicional do método Simplex, implementado de forma sequencial em *CPU* e outra implementação paralela em *GPU CUDA*.

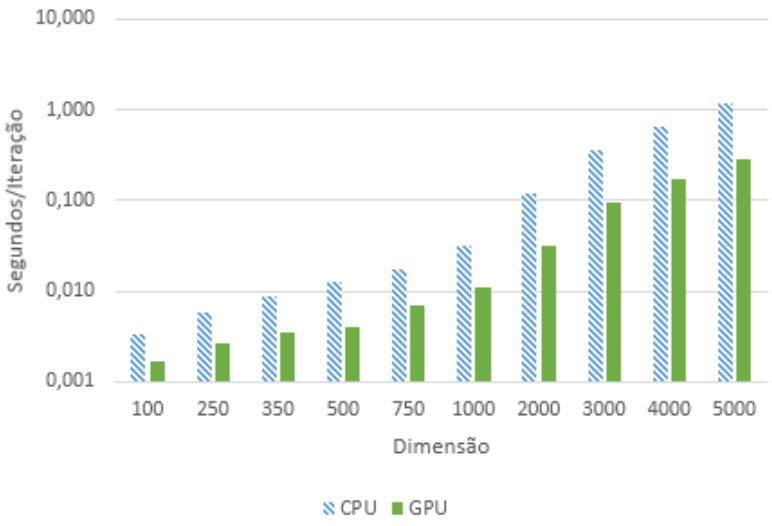

Figura 18 – Desempenho do algoritmo tradicional, em escala logarítmica. Fonte: Do próprio autor.

O speedup alcançado nesta implementação foi de aproximadamente 4,1x. Nota-se que, mesmo a implementação em paralelo utilizando *GPU*, o gráfico apontou uma curva acentuada a medida que o problema de programação linear tem suas dimensões aumentadas, ou seja, a implementação paralela não trouxe um ganho exponencial conforme esperado em relação à implementação sequencial.

Na Fig. 19 mostra o comparativo de desempenho de duas implementações da versão modificada, descrita neste trabalho, do método Simplex.

Diferentemente da implementação anterior, a implementação paralela obteve resultados mais significativos mesmo para problemas com dimensões elevadas, alcançando aproximadamente 9,6x de speedup em relação a implementação sequencial. É muito importante notar que o ganho apresentado neste gráfico nos mostra, além de um ganho superior ao algoritmo tradicional, que a implementação do algoritmo modificado nos trouxe escalabilidade em termos de paralelismo do método Simplex. Tal comportamento não é encontrado na implementação tradicional.

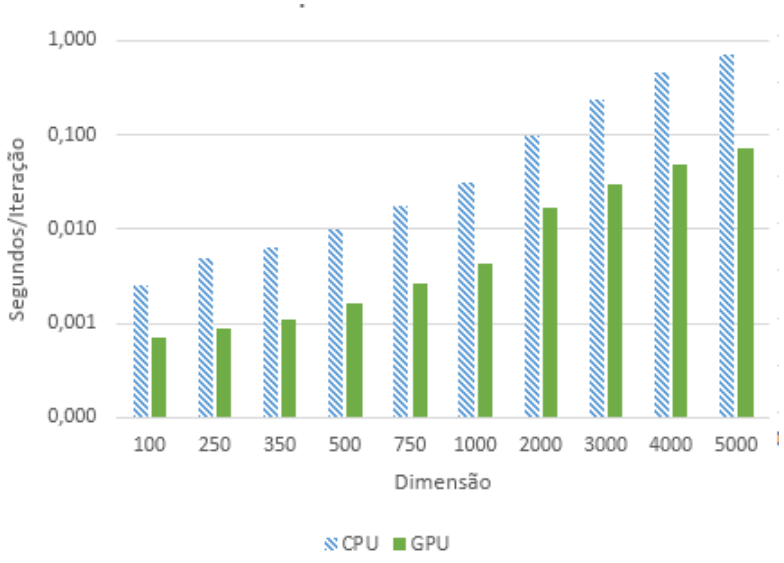

Figura 19 – Desempenho do algoritmo modificado, em escala logarítmica. Fonte: Do próprio autor.

A Fig. 20 nos mostra a comparação entre as implementações do algoritmo sequencial tradicional versus a implementação em paralelo utilizando *CUDA*.

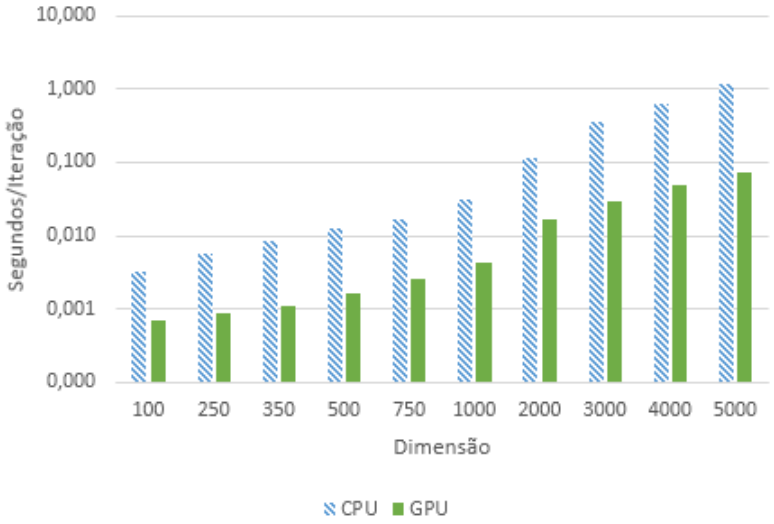

Figura 20 – Desempenho do algoritmo tradicional (CPU) versus algoritmo modificado (GPU), em escala logarítmica. Fonte: Do próprio autor.

Nesta análise observamos um *speedup* de aproximadamente 15,6x entre as implementações do algoritmo. Este tipo de comparação é a mais comum nos trabalhos deste tipo, pois mostra o ganho final total entre uma implementação não otimizada e sequencial em relação a outra implementação otimizada e paralela do algoritmo.

A Fig. 21 nos mostra a relação dos speedups produzidos na comparação mostrada na Fig. 20, indicando uma tendência de aumento do speedup à medida que a dimensão do problema aumenta.

Ainda na Fig. 21, são exibidas duas séries, uma mostrando o *speedup* alcançado utilizando o tempo total do algoritmo em paralelo (tempo de *kernel* mais tempo de transferência de dados), e outra série mostra o *speedup* calculado ao se utilizar apenas o tempo total

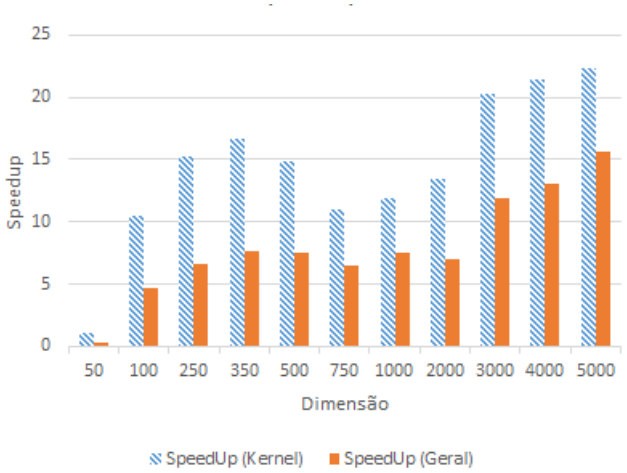

Figura 21 – Speedup alcançado do algoritmo tradicional (*CPU*) versus algoritmo modificado (*GPU*) medido de forma geral (tempo de *kernel* mais tempo de transferência de dados) e medindo apenas o tempo total de *kernel* da *GPU*. Fonte: Do próprio autor.

de processamento dos *kernels* do código paralelo em GPU. Utilizando apenas o tempo de *kernel* como referência, desprezando o tempo de transferência dos dados da matriz entre *host* e *device*, foi possível obter um *speedup* de 22,3x para uma matriz quadrada de ordem 5000 (5000 variáveis por 5000 restrições).

A Fig. 22 mostra um comparativo entre os desempenhos das implementações paralelas utilizando CUDA dos algoritmos tradicional e modificado.

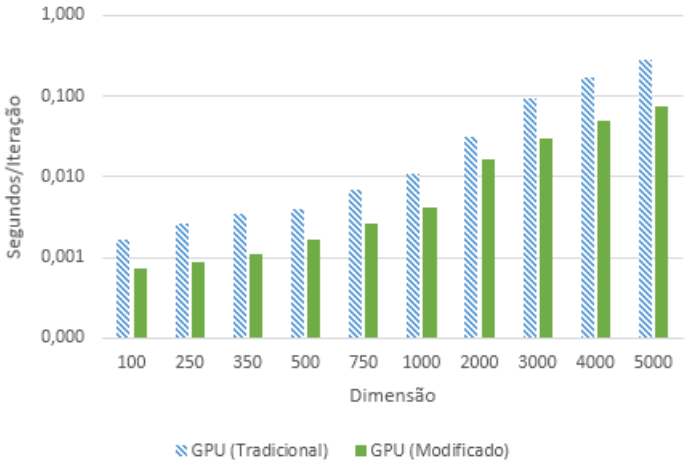

Figura 22 – Desempenho do algoritmo tradicional (GPU) versus algoritmo modificado (GPU), em escala logarítmica. Fonte: Do próprio autor.

O algoritmo modificado apresentou um ganho de desempenho cerca de 3,8x superior ao algoritmo tradicional, demonstrando que a modificação do Simplex apresentada neste trabalho possui um desempenho definitivamente superior à implementação tradicional do algoritmo. Mostrando que a modificação algorítmica proposta, além de produzir ganhos significativos em relação à implementação sequencial, também possui melhor desempenho ao ser paralelizada.

A Tabela 1 nos exemplifica o tempo gasto para encontrar a solução ótima dos problemas. Podemos observar, para o problema de dimensão 5000, que a execução em sequencial

Tabela 1 – DEMONSTRAÇÃO DE TEMPO GASTO PARA ENCONTRAR A SOLUÇÃO ÓTIMA

Dimensão	CPU (segundos)	GPU (segundos)
1000	1586,373	211,225
2000	5837,653	833,653
3000	18093,964	1512,476
4000	32163,231	2469,565
5000	57847,541	3699,255

em *CPU* demoraria mais que 16 horas caso a solução fosse encontrada na 50000° iteração do algoritmo, enquanto o algoritmo em paralelo em GPU demoraria pouco mais de 1 hora para encontrar a mesma solução com o mesmo número de iterações.

6.5 Resultados: Intel Core i7-3615QM 2.30GHz (Octa-Core) vs GPU

Vamos agora analisar os resultados obtidos ao compararmos o desempenho da implementação tradicional e modificada do método Simplex utilizando a *CPU 2* (Intel Core i7):

A Fig. 23 nos mostra a implementação do método Simplex executando de duas formas: uma paralela usando *GPU*, e outra em sequencial usando a *CPU*.

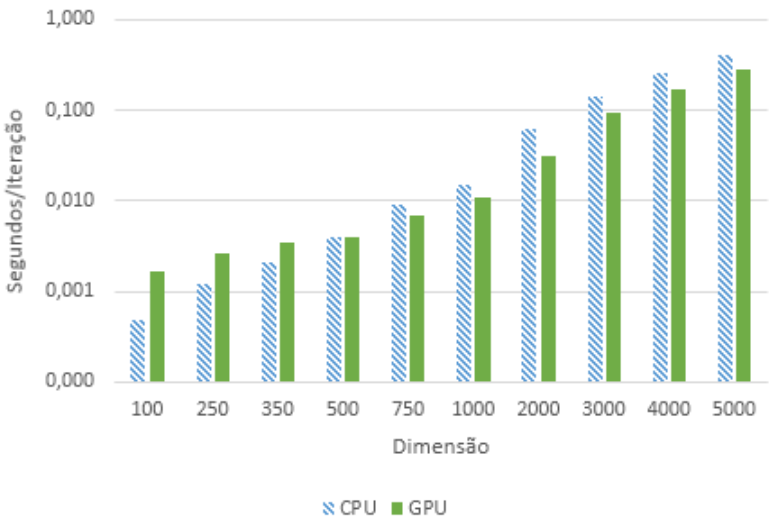

Figura 23 – Desempenho do algoritmo tradicional, em escala logarítmica, implementado usando *GPU* em paralelo versus a implementação sequencial executada numa *CPU*. Fonte: Do próprio autor.

O speedup alcançado nesta implementação foi de aproximadamente 1,4x. Podemos observar neste gráfico que o desempenho entre as implementações não se distanciam consideravelmente, ao passo que a dimensão do problema aumenta. Esse comportamento evidencia que a implementação tradicional do algoritmo não dá resultados significativos mesmo ao ser paralelizada usando *GPU*.

Na Fig. 24 mostra o comparativo de desempenho de duas implementações da versão modificada, descrita neste trabalho, do método Simplex.

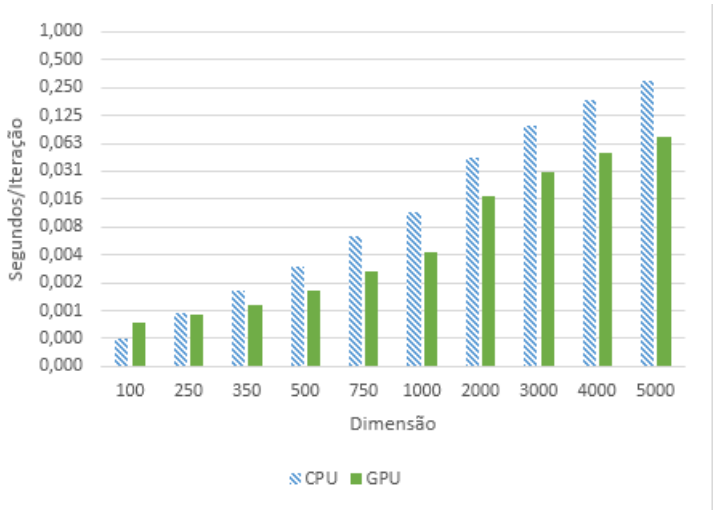

Figura 24 – Desempenho do algoritmo modificado, em escala logarítmica, implementado usando *GPU* em paralelo versus a implementação sequencial executada numa *CPU*. Fonte: Do próprio autor.

Diferente da implementação anterior, a implementação em paralelo obteve resultados mais significativos mesmo para problemas com dimensões elevadas, alcançando aproximadamente 4x de speedup em relação a implementação em sequencial.

A Fig. 25 nos mostra a comparação entre as implementações do algoritmo sequencial tradicional versus a implementação em paralelo utilizando *CUDA*.

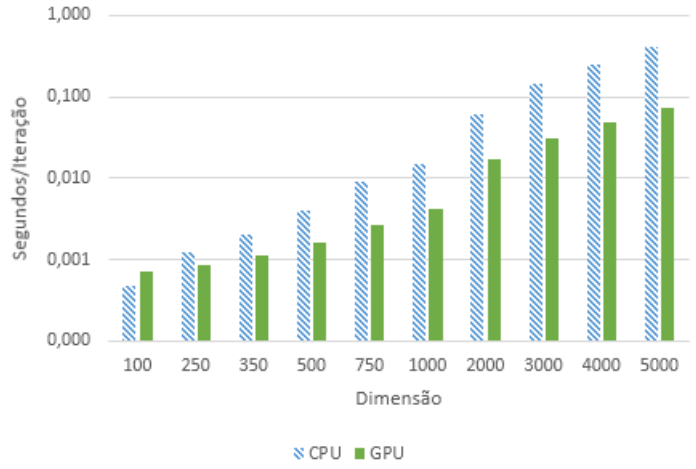

Figura 25 – Desempenho do algoritmo modificado, em escala logarítmica, implementado usando *GPU* em paralelo versus a implementação sequencial do algoritmo tradicional executada em *CPU*. Fonte: Do próprio autor.

Nesta análise observamos um speedup de aproximadamente 5,4x entre as implementações do algoritmo. Baseado neste resultado, observamos que um processador de classe superior ainda consegue ser superado pela implementação paralela do método Simplex, mesmo ao utilizarmos uma *GPU* de classe mais baixa e menor custo.

6.6 Considerações Finais

Neste capítulo apresentamos um comparativo de 4 implementações do método Simplex, foram elas:

- Implementação do método Simplex tradicional implementado de forma sequencial em *CPU*.
- Implementação do método Simplex tradicional implementado de forma paralelo em *CPU-GPU CUDA*.
- Implementação do método Simplex com melhoria algorítmica implementado de forma sequencial em *CPU*.
- Implementação do método Simplex com melhoria algorítmica implementado de forma paralelo em *CPU-GPU CUDA*.

As implementações feitas em sequencial foram testadas com uma *CPU* quad-core e outra *CPU* octa-core, e em todos os casos a implementação feita em paralelo utilizando *GPU CUDA* foi superior nos resultados.

7 CONCLUSÃO

Nesta dissertação, propomos e avaliamos uma implementação paralela do método Simplex em *CUDA*. Nossa implementação obteve melhor desempenho computacional do que as implementações sequencial e paralela do método Simplex tradicional em todos os casos avaliados.

O método Simplex, com a modificação algorítmica proposta, obteve *speedups* consideráveis, de até 15,6x (vide Fig. 20) ao medirmos o tempo total (tempo de transferência mais tempo de processamento de *kernel*), e de até 22,3x (vide Fig. 22) se considerarmos apenas o tempo de processamento total de *kernel*, ao utilizarmos sua implementação em *GPU CUDA* em comparação com sua respectiva implementação em *CPU*. Este ganho de desempenho se mostrou consistente durante os testes, podendo ser superior para matrizes de dimensões ainda maiores.

Salientamos que a modificação algorítmica proposta nos trouxe melhor desempenho tanto quando comparado à implementação sequencial, quanto na implementação paralela do algoritmo tradicional, ou seja, a principal contribuição deste trabalho se dá na apresentação de uma nova modificação algorítmica no método Simplex, que mostrou ser eficiente em termos de desempenho em todos os casos de implementação do método, atingindo o objetivo principal proposto.

Observando os resultados apresentados na Fig. 18, que nos mostra um speedup de 4,1x na implementação do algoritmo tradicional, e na Fig. 19, que nos mostra um speedup de 9,6x na implementação do algoritmo modificado, fica claro que, o algoritmo modificado possui um potencial maior em termos de paralelismo, levando a ganhos superiores e trazendo escalabilidade de desempenho mesmo para problemas grandes.

As implementações geradas durante a pesquisa estão disponíveis em (SILVA, 2015), podendo ser reutilizadas para fins acadêmicos.

7.1 Trabalhos Futuros

Como possíveis trabalhos futuros, poderão ser realizadas melhorias no algoritmo implementado em *GPU*, visando um melhor aproveitamento das características de cada hardware onde a implementação é executada. Essas melhorias dizem respeito à configuração de *grid*, blocos, e quantidade de *threads* cujos *kernels* são submetidos. Esta melhoria pode ser feita utilizando uma estratégia de autoconfiguração do algoritmo, realizada através de uma etapa de reconhecimento de hardware disponível para o processamento.

Outras melhorias ainda podem ser realizadas no que diz respeito à implementação da biblioteca, construindo um módulo *stand-alone* para executar o algoritmo utilizando uma interface gráfica nos moldes da implementação feita pelo próprio autor e disponível no projeto em (SILVA, 2015).

REFERÊNCIAS

BIELING, J.; PESCHLOW, P.; MARTINI, P. An efficient gpu implementation of the revised simplex method. In: **Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on**. [S.l.: s.n.], 2010. p. 1–8.

BOLZ, Jeff et al. Sparse matrix solvers on the gpu: Conjugate gradients and multigrid. In: **ACM SIGGRAPH 2005 Courses**. New York, NY, USA: ACM, 2005. (SIGGRAPH '05). Disponível em: <http://doi.acm.org/10.1145/1198555.1198781>.

DANTZIG, G. B. **Linear Programming and Extensions**. Princeton University Press, 1963. Disponível em: <http://books.google.com.br/books?id=2j46uCX5ZAYC>.

FORREST, Coin|Or John J. **Clp (Coin-or linear programming)**. 2014. Clp (Coin-or linear programming) is an open-source linear programming solver written in C++. It is primarily meant to be used as a callable library, but a basic, stand-alone executable version is also available. Disponível em: <https://projects.coin-or.org/Clp>.

GNU Linear Programming Kit (GLPK). 2000. Disponível em: <http://www.gnu.org/software/glpk/>.

HAMZIC, A.; HUSEINOVIC, A.; NOSOVIC, N. Implementation and performance analysis of the simplex algorithm adapted to run on commodity opencl enabled graphics processors. In: **Information, Communication and Automation Technologies (ICAT), 2011 XXIII International Symposium on**. [S.l.: s.n.], 2011. p. 1–7.

JUNG JIN HYUK, O'leary; P., Dianne. Implementing an interior point method for linear programs on a cpu-gpu system. **ETNA. Electronic Transactions on Numerical Analysis [electronic only]**, Kent State University, Department of Mathematics and Computer Science, v. 28, p. 174–189, 2007. Disponível em: <http://eudml.org/doc/117656>.

KHRONOSGROUP. **Open Computing Language**. 2008. Último acesso em: 01/02/2013. Disponível em: <https://www.khronos.org/opencl/>.

LALAMI, M.E.; BOYER, V.; EL-BAZ, D. Efficient implementation of the simplex method on a cpu-gpu system. In: **Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on**. [S.l.: s.n.], 2011. p. 1999–2006. ISSN 1530-2075.

LALAMI, M.E.; EL-BAZ, D.; BOYER, V. Multi gpu implementation of the simplex algorithm. In: **High Performance Computing and Communications (HPCC), 2011 IEEE 13th International Conference on**. [S.l.: s.n.], 2011. p. 179–186.

MEYER, Paul Albuquerque Xavier; CHOPARD, Bastien. A multi-gpu implementation and performance model for the standard simplex method. **submitted to the 17th International European Conference on Parallel and Distributed Computing**, 2011.

MEYER, X.; CHOPARD, B.; ALBUQUERQUE, P. A branch-and-bound algorithm using multiple gpu-based lp solvers. In: **20th Annual International Conference on High Performance Computing**. [S.l.: s.n.], 2013. p. 129–138. ISSN 1094-7256.

NASH, John C. The (dantzig) simplex method for linear programming. **Computing in Science and Engg.**, IEEE Educational Activities Department, Piscataway, NJ, USA, v. 2, n. 1, p. 29–31, jan. 2000. ISSN 1521-9615. Disponível em: <http://dx.doi.org/10.1109/5992.814654>.

NETLIB. **NETLIB Repository**. 1984. Disponível em: <http://www.netlib.org/>.

NVIDIA, Corporation. **CUDA - CUBLAS Library 2.0**. 2008. Disponível em: <https://developer.nvidia.com/cublas>.

NVIDIA, Corporation. **LAPACK Library**. 2008. Disponível em: <https://developer.nvidia.com/em-photonics-cula-tools>.

PEDRYCZ PETR EKEL, Roberta Parreiras Witold. **Fuzzy Multicriteria Decision-Making: Models, Methods and Applications**. John Wiley & Sons, 2011. 360 p. Disponível em: <http://books.google.com.br/books?id=xjvuNd3wwmsC>.

SILVA, Vinícius Oliveira e. **Cuda Simplex**. 2015. Disponível em: <https://github.com/vinishiru/cuda-simplex>.

SPAMPINATO, D.G.; ELSTER, A.C. Linear optimization on modern gpus. In: **Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on**. [S.l.: s.n.], 2009. p. 1–8. ISSN 1530-2075.

SULLIVAN, Francis; DONGARRA, Jack. Guest editors' introduction: The top 10 algorithms. **Computing in Science & Engineering**, IEEE Computer Society, Los Alamitos, CA, USA, v. 2, n. undefined, p. 22–23, 2000. ISSN 1521-9615.

WIKIPEDIA. **Simplex Algorithm**. 2013. Último acesso: 30 de junho de 2013. Disponível em: <http://en.wikipedia.org/wiki/Simplex_algorithm>.

Wilson, D. G.; Rudin, B. D. Introduction to the ibm optimization subroutine library. **IBM Systems Journal**, v. 31, n. 1, p. 4–10, 1992.