REVISTA DE CIÊNCIAS EXATAS E TECNOLOGIA

SOLUÇÃO DE SISTEMAS LINEARES ESPARSOS UTILIZANDO CUDA
UMA COMPARAÇÃO DE DESEMPENHO EM SISTEMAS WINDOWS E LINUX

Amauri Antunes Filho – *Faculdade Anhanguera de Sorocaba*
Cesar Candido Xavier – *Faculdade Anhanguera de Sorocaba*

RESUMO: Sistemas Lineares Esparsos (SLE) surgem em diversos problemas da engenharia como modelagem atmosférica e simulação de circuitos e normalmente possuem elevado número de incógnitas. Suas soluções requerem elevado esforço computacional e visando minimizar o tempo de solução dessa classe de problemas pode-se utilizar processamento paralelo. Com o avanço tecnológico, a Graphics Processing Unit (GPU) passou a ser utilizada para processamento paralelo, uma vez que esses dispositivos possuem, tipicamente, centenas de núcleos. Foi utilizada a plataforma Compute Unified Device Architecture (CUDA) para implementar a solução de SLE em GPU's da NVIDIA e desenvolvido um programa que resolve SLE utilizando as bibliotecas cusparse e cublas, pertencentes ao CUDA, e comparado o desempenho em dois Sistemas Operacionais (SO): Windows e Linux. Foram realizados testes com seis matrizes esparsas, onde o número de variáveis variou entre 500 K a 1,6 M. Observou-se que a execução no SO Linux é, em média, 5,78% mais eficiente.

PALAVRAS-CHAVE:
cuda; cusparse, cublas, sistemas lineares esparsos.

ABSTRACT: Sparse Linear Systems (SLS) arise in several engineering problems such as atmospheric modeling and circuit simulation and usually has an elevated number of unknowns. Its solutions require a high computational effort and parallel processing can be used to minimize the solution time of this class of problems. The Graphics Processing Unit (GPU) began to be used for parallel processing, once these devices have, typically, hundreds of cores. The Compute Unified Device Architecture (CUDA) platform was used to implement SLS solutions in NVIDIA GPU's. It was developed a computer program that solves SLS using cusparse and cublas libraries. The software's performance was evaluated in two Operating Systems (OS): Windows and Linux. Tests were performed considering six sparse matrices, where the number of variables ranged from 500 K to 1,6 M. It was observed that the execution in Linux OS was, in average, 5,78% more efficient.

KEYWORDS:
cuda; cusparse; cublas; sparse linear systems.

Informe Técnico
Recebido em: 23/06/2013
Avaliado em: 14/02/2014
Publicado em: 17/06/2014

Publicação
Anhanguera Educacional Ltda.

Coordenação
Instituto de Pesquisas Aplicadas e Desenvolvimento Educacional - IPADE

Correspondência
Sistema Anhanguera de Revistas Eletrônicas - SARE
rc.ipade@anhanguera.com

1. INTRODUÇÃO

O poder de processamento de uma *Central Processing Unit* (CPU) deixou de ser suficiente para atender as necessidades de processamento de novas aplicações atualmente. O desenvolvimento de novas gerações de CPU's e o aprimoramento de suas arquiteturas tornou possível o desenvolvimento de softwares com mais funcionalidades, melhor usabilidade e interface com o usuário. Os usuários perceberam esses avanços com naturalidade e demandaram mais melhorias, gerando um ciclo de evolução positivo para a computação (KIRK; HWU, 2010).

Visando aumentar a capacidade de processamento, melhorias foram implementadas na CPU, como o aumento da quantidade de transistores e o aumento da frequência do *clock*. A cada nova geração, estas melhorias foram suficientes para atender às novas necessidades de processamento. A partir de 2003, entretanto, a capacidade de evolução da CPU diminuiu consideravelmente, devido às limitações no processo de fabricação e ao seu consumo de energia e, consequentemente, na necessidade de dissipar o calor da CPU, entre outras (KIRK; HWU, 2010).

Uma solução implementada, visando atender a necessidade de maior capacidade de processamento, foi inserir em um mesmo encapsulamento mais de um núcleo, ou seja, mais de uma CPU. Assim, cada núcleo pode processar informações paralelamente. Várias CPU's em paralelo já eram utilizadas em supercomputadores por várias décadas (SANDERS; KANDROT, 2010). Com isso, a ideia de levar essa solução para os *Personal Computers* (PC), inicialmente com duas CPU's paralelas, tornou-se uma realidade.

A capacidade de processamento paralelo está presente também em uma *Graphics Processing Unit* (GPU) desde o princípio de seu desenvolvimento. As GPU's foram concebidas originalmente para processamento de imagens digitais, especialmente aquelas em três dimensões, utilizadas na indústria do entretenimento, onde são executados milhões de cálculos por segundo (IKEDA, 2011) exigindo, portanto, um elevado poder computacional.

Uma GPU executa milhares de cálculos sobre milhares de dados independentes, ao mesmo tempo, fazendo com que as aplicações executadas por elas pertençam a uma classe diferente das aplicações da CPU tanto em arquitetura quanto em modelo de programação (IKEDA, 2011). A principal diferença da arquitetura entre uma CPU e uma GPU está no fato de que a GPU é desenvolvida com o objetivo de executar o maior número possível de *threads* ao mesmo tempo. O uso da área do *chip* e do consumo de recursos, como energia, para a execução de cálculos também são otimizados em uma GPU (KIRK; HWU, 2010). A CPU, por sua vez, é projetada para a execução mais rápida possível de uma única *thread*, ou seja, é otimizada para a execução de aplicações desenvolvidas de forma sequencial.

O avanço no desenvolvimento das GPU's, tornando-as cada vez mais eficientes e flexíveis, fez com que surgisse uma nova abordagem em seu uso, denominada GPU

Computing. Neste caso o objetivo é utilizar as vantagens das GPU's modernas em aplicações de propósito geral, que sejam paralelizáveis e que façam uso intenso de cálculos (IKEDA, 2011). Diferentes tecnologias para auxiliar no desenvolvimento de softwares que possam ser executados na GPU foram criadas, dentre as quais destaca-se a plataforma *Compute Unified Device Architecture* (CUDA) desenvolvida pela NVIDIA. Essa plataforma disponibiliza recursos para computação paralela, utilizando principalmente a Linguagem C, em três Sistemas Operacionais (SO): Windows, Linux e Mac OS, com suporte para arquiteturas de 32 e 64 bits. É possível utilizá-la em Windows XP, *Vista*, *Seven* e *Eight.* Para Linux há suporte às diferentes distribuições e em diferentes versões e, para o Mac OS, suporte para a versão X.

A plataforma CUDA oferece uma série de recursos que auxiliam no desenvolvimento dos programas. Dentre as diversas ferramentas destacam-se o *NSIGHT* e a CUDA *Libraries.* A primeira funciona como um *plugin* que automatiza a configuração do ambiente de desenvolvimento, realizando as configurações necessárias na *Integrated Development Environment* (IDE), que no caso do Windows é o Visual Studio e no Linux é o Eclipse. A segunda ferramenta é um conjunto de bibliotecas específicas para algumas classes de problemas, como de álgebra linear e manipulação de matrizes esparsas, que neste caso são **cublas** e **cusparse** respectivamente. Estas bibliotecas estão disponíveis na distribuição do CUDA, que atualmente se encontra na versão 5.0.

As soluções de sistemas lineares esparsos podem fazer uso do paralelismo da GPU para otimizar seu processamento. Desta forma é possível encontrar a solução em um menor tempo, quando comparado com a solução em uma CPU. Existem diferentes métodos numéricos para se obter soluções de sistemas lineares esparsos e, consequentemente, diferentes meios de se implementar essas soluções. Porém, dentre os métodos mais utilizados para resolver sistemas de grande escala, que possuem um número muito grande de incógnitas, destacam-se os métodos iterativos e, dentre estes, o método do gradiente conjugado. Existem algoritmos para a implementação desses métodos, sendo necessárias adaptações para que as soluções possam ser obtidas por meio do processamento paralelo realizado na GPU.

2. OBJETIVO

Os objetivos deste trabalho são:

- Descrever sucintamente modelos para armazenamento de matrizes esparsas;
- Apresentar o algoritmo do gradiente conjugado utilizado na solução de sistemas lineares esparsos, destacando os prós e os contras; e
- Expor e discutir os testes de desempenho do programa desenvolvido para solução de sistemas lineares esparsos em dois SO's: Windows e Linux.

3. ORGANIZAÇÃO DO TRABALHO

Este trabalho está organizado da seguinte forma: no Capítulo 2 os sistemas esparsos são caracterizados e suas aplicações e formas de armazenamento são discutidas. Em seguida, no Capítulo 3, é abordado o método do gradiente conjugado para soluções de sistemas lineares esparsos e apresentado o algoritmo para sua implementação. O Capítulo 4 descreve a metodologia utilizada para a realização dos testes de desempenho. Finalmente, o Capítulo 5 apresenta os resultados dos testes, as considerações finais e a conclusão.

4. SISTEMAS ESPARSOS

Sistemas lineares esparsos são sistemas na forma $AX = B$, onde: A é a matriz dos coeficientes das equações; X é o vetor das incógnitas do sistema; e B é o vetor das constantes do sistema. Esses sistemas possuem como característica o fato de que a matriz dos coeficientes A é uma matriz esparsa, ou seja, uma matriz que possui elementos nulos em maior quantidade que elementos não nulos. Um exemplo de matriz esparsa é a matriz tridiagonal, a qual possui elementos não nulos apenas na diagonal principal e nas diagonais acima e abaixo da diagonal principal. A Figura 1 ilustra um exemplo de matriz esparsa porém não tridiagonal

$$A = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Figura 1. Um exemplo de matriz esparsa não tridiagonal.

As matrizes de coeficientes dos sistemas lineares esparsos utilizados como referência nesse trabalho possuem três caraterísticas em comum: são quadradas; simétricas; e positivas definidas. A seguir, define-se cada uma destas características.

Uma matriz quadrada é aquela que possui o mesmo número de linhas e colunas, ou seja, uma matriz A de ordem m x n é quadrada se $m = n$, onde n é o número de linhas e m é o número de colunas. Neste caso, diz-se que a matriz é de ordem n. Quando a matriz dos coeficientes é quadrada, o sistema possui um número igual de equações e de incógnitas.

Diz-se que uma matriz é simétrica quando ela for quadrada e igual a sua transposta. Denomina-se a transposta da matriz A como sendo A^T. Seja A uma matriz de ordem m x n. A^T será uma matriz de ordem n x m. Seja a_{ij} um elemento da matriz A, onde $1 \leq i \leq m$ e $1 \leq j \leq n$. Os elementos de A^T, a^T_{ji}, serão $a^T_{ji} = a_{ij}$. Ou seja, a transposta de uma matriz A é obtida pela permutação das linhas pelas colunas de mesmo índice.

Para que uma matriz A de ordem n seja definida positiva, ela precisa atender ao seguinte requisito dado um vetor z não nulo: $z^T Az > 0$, para qualquer z.

Sistemas lineares podem ser resolvidos utilizando métodos iterativos ou diretos (NAUMOV, 2011). Como são, tipicamente, originados da engenharia e podem ser resolvidos iterativamente, as soluções desses tipos de sistemas normalmente são automatizadas por meio do uso de sistemas computacionais.

Tipicamente sistemas esparsos possuem um número muito grande de equações e incógnitas inviabilizando a resolução manual e o armazenamento na memória do computador das matrizes que representam esses sistemas. Por isso foram desenvolvidas técnicas que otimizam o uso de memória disponível para o armazenamento dessas matrizes e solução destes sistemas lineares.

No que diz respeito ao armazenamento dessas matrizes observa-se que as propostas são baseadas no armazenamento apenas dos valores não nulos, que são relevantes para a solução do sistema.

A Universidade da Flórida (FLÓRIDA, 2013), nos Estados Unidos, possui um acervo de matrizes esparsas de diversos tipos, incluindo as matrizes simétricas positivas definidas ou, do inglês, *symmetric positive definite* (SPD). Essas matrizes estão disponíveis em arquivos para *downloads*. Um dos formatos de arquivo disponíveis para *download* é o *Matrix Market Format* (MMF), cujo formato é específico para o armazenamento de matrizes. As matrizes utilizadas nos testes desse trabalho foram obtidas a partir desse acervo e estão relacionadas a diferentes áreas de conhecimento.

A seguir serão descritos os principais formatos de armazenamento de uma matriz esparsa na memória do computador.

4.1. Matriz no Formato Denso

O formato denso, ou *Dense Format*, é um formato de armazenamento comum de matrizes. Nesse formato são armazenados todos os valores não nulos e nulos na memória do computador.

Este fator faz com que os requisitos de memória para manipulação dessas matrizes sejam muito elevados. As Figuras 2 e 3 mostram um exemplo de matriz e sua forma de armazenamento nesse formato respectivamente.

$$\begin{bmatrix} X_{1,1} & X_{1,2} & \cdots & X_{1,n} \\ X_{2,1} & X_{2,2} & \cdots & X_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ X_{m,1} & X_{m,2} & \cdots & X_{m,n} \\ \vdots & \vdots & \ddots & \vdots \\ X_{ldX,1} & X_{ldX,2} & \cdots & X_{ldX,n} \end{bmatrix}$$

Figura 2. Representação de uma matriz no formato denso.

$$\begin{bmatrix} X_{1,1} & X_{2,1} & \cdots & X_{m,1} & \cdots & X_{ldX,1} & \cdots & X_{1,n} & X_{2,n} & \cdots & X_{m,n} & \cdots & X_{ldX,n} \end{bmatrix}$$

Figura 3. Representação da matriz no formato denso armazenada na memória do computador.

4.2. Matriz no Formato Coordenado (COO)

O *Coordinate Format* (COO), ou formato coordenado, é um método de armazenamento de matrizes esparsas. Neste caso são utilizados três vetores para o armazenamento dos valores não nulos da matriz: cooValA; cooRowIndA; e cooColIndA. O vetor cooValA armazena os valores não nulos da matriz sequencialmente por ordem de linha. Os vetores cooRowIndA e cooColIndA armazenam os índices referentes às linhas e colunas em que os elementos não nulos estão posicionados respectivamente. As Figuras 4 e 5 apresentam uma matriz e seu respectivo modelo de armazenamento no formato COO.

$$\begin{bmatrix} 1.0 & 4.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 2.0 & 3.0 & 0.0 & 0.0 \\ 5.0 & 0.0 & 0.0 & 7.0 & 8.0 \\ 0.0 & 0.0 & 9.0 & 0.0 & 6.0 \end{bmatrix}$$

Figura 4. Representação da matriz que deverá ser armazenada no formato COO.

$$\begin{aligned} \text{cooValA} &= [1.0\ \ 4.0\ \ 2.0\ \ 3.0\ \ 5.0\ \ 7.0\ \ 8.0\ \ 9.0\ \ 6.0] \\ \text{cooRowIndA} &= [0\ \ 0\ \ 1\ \ 1\ \ 2\ \ 2\ \ 2\ \ 3\ \ 3\ ] \\ \text{cooColIndA} &= [0\ \ 1\ \ 1\ \ 2\ \ 0\ \ 3\ \ 4\ \ 2\ \ 4\ ] \end{aligned}$$

Figura 5. Matriz da Figura 4 armazenada no formato COO.

4.3. Matriz no Formato de Linha Comprimida (CSR)

O *Compressed Sparse Row* (CSR) ou linha esparsa comprimida é um método de armazenamento de matrizes esparsas que, assim como o formato COO, utiliza três vetores para armazenar os valores não nulos da matriz assim denominados: csrValA, csrRowPtrA e csrColIndA. O vetor csrValA armazena os valores não nulos da matriz, o vetor csrRowPtrA armazena os índices identificando a posição no vetor csrValA onde se inicia uma nova linha da matriz e o vetor csrColIndA armazena os índices referentes às colunas em que os elementos não nulos estão posicionados na matriz.

A Figura 6 ilustra a matriz da Figura 4 armazenada no formato CSR.

$$\begin{aligned} \text{csrValA} &= [1.0\ \ 4.0\ \ 2.0\ \ 3.0\ \ 5.0\ \ 7.0\ \ 8.0\ \ 9.0\ \ 6.0] \\ \text{csrRowPtrA} &= [0\ \ 2\ \ 4\ \ 7\ \ 9\ ] \\ \text{csrColIndA} &= [0\ \ 1\ \ 1\ \ 2\ \ 0\ \ 3\ \ 4\ \ 2\ \ 4\ ] \end{aligned}$$

Figura 6. Matriz da Figura 4 armazenada no formato CSR.

4.4. Matriz no Formato de Coluna Comprimida (CSC)

O formato *Compressed Sparse Column* (CSC) ou coluna esparsa comprimida é um formato de

armazenamento de matrizes esparsas que utiliza, assim como nos dois formatos descritos anteriormente, três vetores para o armazenamento dos valores não nulos da matriz: cscValA; cscRowIndA; e cscColPtrA. O vetor cscValA armazena os valores não nulos da matriz, o vetor cscRowIndA armazena os índices referentes às linhas em que os elementos não nulos da matriz estão posicionados e o vetor cscColPtrA armazena os índices identificando a posição no vetor cscValA onde se inicia uma nova coluna da matriz.

A Figura 7 ilustra a matriz da Figura 4 armazenada no formato CSC.

```
cscValA    = [1.0 5.0 4.0 2.0 3.0 9.0 7.0 8.0 6.0]
cscRowIndA = [0   2   0   1   1   3   2   2   3 ]
cscColPtrA = [0   2   4   6   7   9 ]
```

Figura 7. Matriz da Figura 4 armazenada no formato CSC.

4.5. Matrix Market Format

O *Matrix Market Format* (MARKET, 2013) é um formato de arquivo onde são armazenados os valores não nulos de matrizes esparsas. Os arquivos nesse formato possuem uma estrutura própria, e podem armazenar diferentes tipos de matrizes esparsas e vetores.

Os arquivos possuem um cabeçalho que contém as informações para a leitura do arquivo. As informações contidas no cabeçalho são: o tipo de arquivo; se o arquivo representa uma matriz ou um vetor; qual o formato que a matriz está armazenada; o tipo de dados contido na matriz; e, finalmente, o tipo de matriz.

Geralmente o tipo de arquivo é incluído para o caso em que uma aplicação que realize a leitura de um arquivo *Matrix Market Format* (MMF) possa distingui-lo de outros tipos de arquivos, como o MATLAB *Format* e o Rutherford/Boeing *Format*. O formado da matriz pode ser *Coordinate Format* ou *Array Format*. O *Coordinate Format* informa que a matriz ali armazenada está no formato COO e o *Array Format* indica que a matriz está armazenada no formato denso. Os tipos de dados são referentes aos valores não nulos armazenados na matriz, os quais podem ser reais ou complexos. Neste trabalho todas as matrizes utilizadas possuem o tipo de dado *real*. Finalmente, o tipo de matriz é referente à estrutura da matriz os quais podem ser: *general*; *symmetric*; *sknew-symmetric*; ou *hermitan*. As matrizes aqui utilizadas, são do tipo *symmetric* e, neste caso, além do cabeçalho, se armazena apenas a parte inferior da matriz, que inclui a diagonal principal e os valores abaixo dela.

O arquivo no formato MMF pode possuir também linhas de comentários antes das linhas contendo os valores não nulos da matriz e seus respectivos índices. Uma linha de comentário deve obrigatoriamente iniciar com o caractere "%".

A primeira linha após os comentários possui três parâmetros: dois números inteiros separados por um espaço em branco, indicando a quantidade de linhas e de colunas; e um

terceiro valor inteiro, também separado por um espaço em branco, que informa o número de elementos não nulos da matriz.

Vale destacar que, para o caso de arquivos que armazenam matrizes simétricas, o número de elementos não nulos informado é referente somente à quantidade de números armazenados no arquivo e não ao existente na matriz. Neste caso o arquivo contém os valores da diagonal principal da matriz e os valores não nulos abaixo da diagonal principal.

A Figura 8 ilustra um exemplo de arquivo no formato MMF de uma matriz 5 x 5, não simétrica, com 8 elementos não nulos.

```
%%MatrixMarket matrix coordinate real general
%=================================================================================
%
% This ASCII file represents a sparse MxN matrix with L
% nonzeros in the following Matrix Market format:
%
% +----------------------------------------------+
% |%%MatrixMarket matrix coordinate real general | <--- header line
% |%                                             | <--+
% |% comments                                    |    |-- 0 or more comment lines
% |%                                             | <--+
% |    M  N  L                                   | <--- rows, columns, entries
% |    I1  J1  A(I1, J1)                         | <--+
% |    I2  J2  A(I2, J2)                         |    |
% |    I3  J3  A(I3, J3)                         |    |-- L lines
% |        . . .                                 |    |
% |    IL JL  A(IL, JL)                          | <--+
% +----------------------------------------------+
%
% Indices are 1-based, i.e. A(1,1) is the first element.
%
%=================================================================================
  5  5  8
    1     1   1.000e+00
    2     2   1.050e+01
    3     3   1.500e-02
    1     4   6.000e+00
    4     2   2.505e+02
    4     4  -2.800e+02
    4     5   3.332e+01
    5     5   1.200e+01
```

Figura 8. Exemplo de arquivo Matrix Market Format.

Algoritmo para Leitura de um Arquivo MMF

Para a utilização das ferramentas providas pelo CUDA, é necessário armazenar em memória física todos os valores da matriz. A simples leitura do arquivo MMF carregando os valores em vetores não é suficiente quando se trata de arquivos contendo matrizes simétricas.

Por isso foi necessário o desenvolvimento de um algoritmo para realizar a leitura dos dados do arquivo, identificando os valores com índices abaixo da diagonal principal e criando índices acima da diagonal principal. Todos esses valores devem, em seguida, ser armazenados sequencialmente em três vetores no formato COO, para que, posteriormente, possa ser utilizada a função de mudança do formato de armazenamento da matriz, contida na biblioteca cusparse e abordada mais adiante. A conversão deve ser feita devido ao fato de que as demais funções utilizadas para a solução do sistema necessitam que a matriz esteja armazenada no formato CSC.

Inicialmente é realizada a leitura do arquivo e o armazenamento dos valores não nulos da matriz, exatamente como no arquivo, em três vetores na memória com o tamanho igual ao de valores não nulos no formato de armazenamento COO. Em seguida são criados outros três vetores, também no formato COO, porém com tamanho calculado para se armazenar

também os valores acima da diagonal principal, não inseridos no arquivo por se tratar de uma matriz simétrica, para armazenar os valores da matriz completa. A Figura 9 mostra o algoritmo usado para gerar a parte acima da diagonal principal e juntar os valores gerados com os valores obtidos do arquivo, uma vez que a diagonal principal e os valores abaixo já estão contidos no arquivo.

```
1  /*
2      OS VETORES I_inf[], J_inf[] E val_fin[] CONTÉM OS VALORES,
3      ORIGINADOS DO ARQUIVO EM MMF OBTIDO DO REPOSITÓRIO DA
4      UNIVERSIDADE DA FLÓRIDA, EQUIVALENTES AOS VETORES cooRowIndA,
5      cooColIndA E cooValA, RESPECTIVAMENTE, REPRESENTADOS NA FIGURA 5.
6      OS VETORES I_fin, J_fin E val_fin ARMAZENARÃO OS VALORES DA MATRIZ
7      COMPLETA, OBTIDOS APÓS A EXECUÇÃO DO ALGORITMO.
8  */
9
10 inteiro posicao <- 0
11 inteiro linha <- 0
12 inteiro cont <- 0
13 inteiro nz <- quantidade de valores não nulos da matriz, originado do arquivo
14 inteiro N <- quantidade de linhas na matriz, originado do arquivo
15
16 para linha de 0 até N - 1 passo 1 faça
17    para cont de 0 até nz passo 1 faça
18        se (I_inf[cont] = linha) então
19            início
20                I_fin[posicao] <- I_inf[cont];
21                J_fin[posicao] <- J_inf[cont];
22                val_fin[posicao] <- val_inf[cont];
23                posicao <- posicao + 1;
24            fim
25        fim se
26    fim para
27
28    para cont de 0 até nz passo 1 faça
29        se (I_inf[cont] = linha) então
30            início
31                I_fin[posicao] <- J_inf[cont];
32                J_fin[posicao] <- I_inf[cont];
33                val_fin[posicao] <- val_inf[cont];
34                posicao <- posicao + 1;
35            fim
36        fim se
37    fim para
38 fim para
```

Figura 9. Algoritmo utilizado para geração e armazenamento, junto com os dados previamente obtidos do arquivo do repositório da Universidade da Flórida, da parte superior de uma matriz simétrica a partir do arquivo em MMF.

As linhas de 9 a 13 inicializam as variáveis utilizadas no algoritmo. O laço iniciado na linha 15 e encerrado na linha 37 executa os laços internos por N vezes. O laço das linhas 16 a 25 é responsável por colocar os valores da diagonal principal e dos valores abaixo dela na matriz final. O laço das linhas 27 a 36 é responsável por colocar os valores acima da diagonal principal na matriz final. Essa inclusão dos valores é possível invertendo-se os valores dos índices de linha e coluna obtidos a partir da matriz no fomato MMF.

Após obter a matriz completa no formato COO deve-se convertê-la ao formato CSR. Esta conversão deve-se ao fato de que as funções da biblioteca cusparse do CUDA exigem

que a matriz passada como parâmetro esteja nesse formato de armazenamento. A biblioteca cusparse e a metodologia utilizada à conversão entre os formatos de armazenamento das matrizes serão abordados no Capítulo 4.

5. MÉTODO DO GRADIENTE CONJUGADO

O método iterativo mais utilizado para solução de sistemas lineares com um número elevado de incógnitas é o Método do Gradiente Conjugado (MGC) (SHEWCHUK, 1994). A matriz dos sistemas que utilizam esse método são geralmente quadradas, simétricas e positivas definidas (SHEWCHUK, 1994).

O principal atrativo no uso de métodos iterativos para a solução de sistemas esparsos é o fato de que este procedimento realiza sobre a matriz A apenas duas operações: multiplicação de A e A^T por um vetor (PRESS et al., 1992).

O MGC busca a solução do sistema através da minimização do erro obtido a partir da solução encontrada em cada iteração. Para isso é utilizado um vetor solução inicial e, a partir daí, o algoritmo busca se aproximar da solução gerando um novo vetor solução. Define-se um parâmetro de erro aceitável e, quando o vetor solução encontrado em determinado passo de iteração é inferior a esse parâmetro, o algoritmo é encerrado. Para todas as matrizes estudas, antes de executar o algoritmo do MCG, o vetor solução foi inicializado com o valor 0 em todas as posições.

Existem variações do MGC devido a diferentes propriedades da matriz A do sistema. Quando essa matriz é simétrica e positiva definida, utiliza-se a forma básica do algoritmo. O estudo aqui realizado foi baseado em sistemas nos quais as matrizes são simétricas e positivas definidas e, por isso, o algoritmo aqui apresentado resolve apenas esses tipos de sistemas e está representado na Figura 10.

O algoritmo utiliza a matriz dos coeficientes A, os vetores: b com as constantes do sistema; x com a solução do sistema e que é atualizado a cada iteração; e x_aux utilizado como um vetor de solução temporário. Também são utilizados valores reais *alfa*, *beta*, *p_aux*, *r* e *tolerancia*, além do valor inteiro k. Essas variáveis são inicializadas pelo algoritmo nas linhas de 18 a 21.

```
1  /*
2      A MATRIZ DE REAIS "A" ARMAZENARÁ OS VALORES DA MATRIZ DE
3      COEFICIENTES DO SISTEMA. OS VALORES alfa E beta SÃO
4      OBTIDOS A CADA ITERAÇÃO DA SOLUÇÃO. OS VETORES b E
5      x REPRESENTAM O VETOR CONSTANTE DO SISTEMA E O VETOR
6      SOLUÇÃO RESPECTIVAMENTE. O REAL r É O RESÍDUO OBTIDO
7      EM CADA INTERAÇÃO. O VETOR x_aux E OS REAIS p_aux e r_aux
8      SÃO VALORES TEMPORÁRIOS UTILIZADOS A CADA PASSO. O
9      INTEIRO k CONTROLA QUANTAS ITERAÇÕES FORAM NECESSÁRIAS PARA
10     SE ENCONTRAR A SOLUÇÃO.
11 */
12
13 matriz de reais A
14 vetor de reais b, x, x_aux
15 inteiro k
16 real alfa, beta, p_aux, p, r_aux, r, tolerancia
17
18 x <- vetor solução suposto inicialmente;
19 r <- b - A * x;
20 k <- 0;
21 tolerancia <- valor aceito como tolerância de erro para a solução;
22
23 enquanto (r > tolerancia*tolerancia) faça
24
25    alfa <- (r * r) / (p * A * p);
26    r_aux <- r - (alfa * A * p);
27    beta <- (r_aux * r_aux) / (r * r);
28    p_aux <- r_aux + (beta * p);
29
30    x_aux <- x + (alfa * p);
31
32    r <- r_aux;
33    p <- p_aux;
34    x <- x_aux;
35    k <- k + 1;
36
37 fim enquanto;
```

Figura 10. Algoritmo do método do Gradiente Conjugado.

O laço entre as linhas 23, a 37 é responsável por, iterativamente, se aproximar da solução do sistema. A cada iteração é calculado *r*, que é o resíduo de erro da solução. Se o valor do resíduo for menor que a tolerância, considera-se que a solução foi encontrada e o laço é finalizado.

Em cada iteração são calculados os valores de *alfa* e *beta*, os quais são utilizados para se determinar os novos valores do vetor solução *x*. É calculado o resíduo de erro e incrementado o inteiro *k*, responsável por armazenar, no final da execução do algoritmo, o número de iterações utilizado para se encontrar a solução do sistema.

6. METODOLOGIA

O trabalho baseou-se no desenvolvimento de um programa em C capaz de resolver sistemas lineares esparsos, utilizando a GPU. Também foram realizados os testes de desempenho do programa desenvolvido em dois SO's: Windows e Linux.

As principais caraterísticas do computador utilizado neste trabalho são:

- Processador Intel Core i7 – 2600K 3.40 GHz;
- Placa Mãe ASUS P8Z68 V LX;
- Placa de Vídeo NVIDIA 2GB GTX 560 TI;
- Memória 8GB 1333 KINGSTON (2X4GB);
- Windows 7 Professional; e
- Ubuntu 11.10;

A placa de vídeo GTX 560 TI possui as seguintes características:

- Número de Núcleos: 384;
- Clock Placa Gráfica (MHz): 822;
- Clock do Processador (MHz): 1645;
- Clock da Memória (Gbps): 4008;
- Memória: 2048 MB GDDR5;
- Largura (bits) da Interface com a Memória: 256; e
- Largura de banda da Memória (GB/sec): 128.

Os SO's foram instalados em *Dual Boot* onde, no mesmo computador, pode-se utilizar mais de um SO selecionando qual se deseja utilizar no momento da inicialização. Para a utilização do CUDA é necessário que a GPU possua suporte à plataforma. Este suporte pode ser verificado consultando o site da NVIDIA (CUDA GPU's, 2013).

A instalação do CUDA *Toolkit* e do CUDA *Driver* pode ser realizada por meio de download no site da NVIDIA (CUDA, 2013).

O programa desenvolvido para determinar a solução de sistemas lineares esparsos foi desenvolvido a partir de um exemplo, disponível no sítio (CUDA SAMPLES, 2013), que implementa o método GC utilizando as bibliotecas cusparse e cublas, que serão abordadas mais adiante. Foi necessário adaptar o código para que a matriz não fosse gerada aleatoriamente, mas sim a partir do repositório de matrizes da Universidade da Flórida (FLÓRIDA, 2013).

Foram utilizadas seis matrizes para a execução dos testes. A Tabela 1 enumera as matrizes utilizadas neste trabalho, a quantidade de linhas (N) e a quantidade de valores não nulos (nnz).

Tabela 1. Relação de matrizes utilizadas nos testes.

	Matriz	N	nnz
1	af_shell3	504.855	17.562.051
2	parabolic_fem	525.825	3.674.625
3	apache2	715.176	4.817.870
4	ecology2	999.999	4.995.991
5	thermal2	1.228.045	8.580.313
6	G3_circuit	1.585.478	7.660.826

Para cada matriz foram realizadas dez execuções de teste e, para cada execução, foram medidos os tempos de três tarefas na execução: cópia da matriz para a memória da GPU; solução do sistema; e a cópia de volta para a memória do computador. Ao final, as três medidas de tempo foram somadas e assim obtido o tempo total de execução pela GPU.

A seguir são abordadas as características das bibliotecas do CUDA utilizadas para a implementação da solução dos sistemas lineares esparsos.

6.1. CUSPARSE

A biblioteca cusparse possui um conjunto de sub-rotinas de álgebra linear que operam sobre as matrizes esparsas, que podem ser utilizadas em programas escritos em linguagem C e C++ (CUSPARSE, 2013).

Essa biblioteca possui quatro classificações para conjuntos de sub-rotinas denominadas *Level* 1, *Level* 2, *Level* 3 e de conversão. Cada categoria possui um conjunto de sub-rotinas com características em comum, como descrito a seguir:

- *Level* 1: realiza operações entre dois vetores sendo um no formato esparso e outro no formato denso;
- *Level* 2: realiza operações entre uma matriz esparsa e um vetor denso;
- *Level* 3: realiza operações entre uma matriz esparsa e um conjunto de vetores densos; e
- Conversão: realiza conversão entre os diferentes formatos de matrizes, como por exemplo do formato COO para o formato CSR.

A matriz obtida a partir arquivo do MMF é armazenada no formato COO porém, para as operações da biblioteca cusparse que necessitam ser utilizadas para a implementação da solução dos sistemas, é necessário que a matriz esteja no formato CSR.

6.2. CUBLAS

A biblioteca cublas possui um conjunto de sub-rotinas básicas de álgebra linear. Para sua utilização é necessário que seja alocada memória para os dados das matrizes e dos vetores na GPU. Essa biblioteca também suporta múltiplas GPU's, embora não faça a distribuição de processamento de maneira automática, cabendo ao programador implementar esse recurso.

Esta biblioteca possui também conjuntos de sub-rotinas classificadas como *Level* 1, *Level* 2 e *Level* 3. As sub-rotinas de *Level* 1 são utilizadas para operações utilizando vetores; as sub-rotinas de *Level* 2 são utilizadas para operações entre matrizes e vetores; e as sub-rotinas de *Level* 3 são utilizadas para operações entre matrizes.

7. RESULTADOS E CONCLUSÕES

Foram realizadas dez execuções para cada sistema listado na Tabela 1, em cada SO. Como tolerância de erro para o MGC foi utilizado o valor de 1e-5.

A Tabela 2 relaciona as matrizes utilizadas e as médias de tempo de suas execuções no SO Windows onde todos os valores estão em segundos, além do número de iterações necessárias até se encontrar a solução do sistema.

Tabela 2. Média dos tempos de execução no SO Windows.

Matriz	Cópia para GPU	Solução	Cópia para computador	Total	Número de iterações
af_shell3	0,0564929	18,8584336	0,0010109	18,915937	4087
parabolic_fem	0,0149009	2,9668496	0,0010796	2,9828299	1657
apache2	0,0190188	10,3920854	0,0014226	10,4125268	4777
ecology2	0,0210434	15,7176283	0,001993	15,7406638	6190
thermal2	0,0319775	21,5158552	0,00240564	21,5502627	5511
G3_circuit	0,031383	66,219343	0,0035886	66,2543142	16993

A Tabela 3 relaciona as matrizes e as médias de tempo de suas execuções no SO Linux, com valores de tempo de execução também em segundos e a quantidade de iterações necessárias até se encontrar a solução do sistema.

Tabela 3. Média dos tempos de execução no SO Linux.

Matriz	Cópia para GPU	Solução	Cópia para computador	Total	Número de iterações
af_shell3	0,0389264	17,8244354	0,0008078	17,8641692	4091
parabolic_fem	0,0106026	2,7337032	0,0008317	2,7451376	1657
apache2	0,0136391	9,7140877	0,0010857	9,7288127	4777
ecology2	0,014981	14,8579087	0,0014731	14,8743627	6190
thermal2	0,0230007	20,7560659	0,0017871	20,7808544	5511
G3_circuit	0,0224214	63,9557316	0,0022697	63,9804227	16993

Em média, o SO Linux foi 5,78% mais eficiente no tempo total de execução do que o Windows. Esta eficiência variou de 3,55% para a matriz G3_circuit até 8,66% para a matriz parabolic_fem.

Pode-se observar que a maior diferença de desempenho entre os dois SO foi na cópia dos dados da matriz da memória do computador para a memória da GPU e, em seguida, na cópia dos dados da GPU para o computador. O tempo de cópia, no SO Linux, de dados para a GPU e vice-versa foram 40,76% e 35,67%, em média e respectivamente, menores quando comparados com o SO Windows.

Embora a transferência dos dados da memória do computador para a memória da GPU e depois de volta para o computador seja a etapa com maior diferença de desempenho, essas etapas representam, respectivamente, 0,22% e 0,01%, em média, do tempo total de execução. A etapa de solução do sistema é, em média, responsável por 99,77% do tempo total de execução.

Portanto, observando os resultados dos testes pode-se afirmar, para os sistemas estudados, que independente da configuração do hardware, a tarefa de cópia de dados entre as memórias, do computador e da GPU e a determinação da solução de sistemas lineares esparsos é melhor desempenhada pelo SO Linux.

Também pode-se constatar que, a matriz af_shell3 possui quantidades diferentes de iterações em cada SO e, o motivo desta diferença, será objeto de investigação futura.

REFERÊNCIAS

CUBLAS. Disponível em: <http://docs.nvidia.com/cuda/cublas/index.html>. Acesso em: 16 mar. 2013.

CUDA. Título Disponível em: <https://developer.nvidia.com/cuda-downloads>. Acesso em: 16 mar. 2013.

CUDA GPU's. Disponível em: <https://developer.nvidia.com/cuda-gpus>. Acesso em: 16 mar. 2013.

CUDA SAMPLES. Disponível em: <http://docs.nvidia.com/cuda/cuda-samples/index.html>. Acesso em: 16 mar. 2013.

CUSPARSE. Título Disponível em: <http://docs.nvidia.com/cuda/cusparse/index.html>. Acesso em: 16 mar. 2013.

FLÓRIDA, University Of. The University of Florida Sparse Matrix Collection. Disponível em: <http://www.cise.ufl.edu/research/sparse/matrices/>. Acesso em: 07 mar. 2013.

IKEDA, Patricia Akemi. Um estudo do uso eficiente de programas em placas gráficas. 2011. 80 p. Dissertação (Mestrado em Ciências da Computação)- Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2011.

KIRK, David B.; HWU, Wen-mei W. Programming Massively Parallel Processors: A Hands-on Approach. 1. ed. [S.l.]: Elsevier Science, 2010. 280 p.

MARKET, Matrix. Matrix Market Exchange Formats. Disponível em: <http://math.nist.gov/MatrixMarket/formats.html#MMformat>. Acesso em: 07 mar. 2013.

NAUMOV, Maxim. Incomplete-LU and Cholesky Preconditioned Iterative Methods Using CUSPARSE and CUBLAS. California: [s.n.], 2011. 16 p. Disponível em: <https://developer.nvidia.com/content/incomplete-lu-and-cholesky-preconditioned-iterative-methods-using-cusparse-and-cublas>. Acesso em: 07 mar. 2013.

PRESS, William H. et al. Numerical Recipes in C: The Art of Scientific Computing. 2. ed. Cambridge: Cambridge University Press, 1992. 965 p. Disponível em: <http://apps.nrbook.com/c/index.html>. Acesso em: 07 mar. 2013.

SANDERS, Jason; KANDROT, Edward. CUDA by Example: An Introduction to General-Purpose GPU Programming. 1. ed. Michigan: NVIDIA, 2010. 289 p.

SHEWCHUK, Jonathan Richard. An Introduction to the Conjugate Gradient Method Without the Agonizing Pain. 1. ed. Pittsburgh: [s.n.], 1994. 64 p.

STEINBRUCH, Alfredo; WINTERLE, Paulo. Álgebra Linear e Geometria Analítica. 2. ed. São Paulo: Pearson, 1987. 469 p.