ANÁLISE NODAL DE CIRCUITO ELÉTRICO EMPREGANDO COMPUTAÇÃO PARALELA GPU CUDA

Machado, B.S.[1,2], Chaves, L. B.[3] Galvão Filho, A.R. [1,2], Carvalho, R.V. [1,2], Coelho, C.J. [1,2]

[1] Laboratório de Computação Científica - Escola de Ciências Exatas e da Computação

Pontifícia Universidade Católica de Goiás - Goiânia-Goiás-Brasil

[2] Mestrado em Engenharia de Produção e Sistemas

Pontifícia Universidade Católica de Goiás - Goiânia-Goiás-Brasil

[3]Núcleo de Desenvolvimento de Software - Fundação Aroeira - São Paulo – Brasil

RESUMO: Este trabalho apresenta uma abordagem de processamento paralelo para a solução de sistemas lineares que representam nós de circuito elétrico para o cálculo de suas tensões e demais grandezas elétricas: corrente e potência. A análise nodal é baseada na primeira Lei dos nós de Kirchhoff e é usada para a resolução do circuito elétrico através de equações que formam um sistema linear. A solução deste sistema linear é obtida com o processamento paralelo empregando o hardware da placa de vídeo e a arquitetura de dispositivo de computação unificada. Os resultados obtidos para os circuitos de teste é de 56,54%. É possível concluir que a abordagem paralela é uma boa alternativa para reduzir o tempo de processamento de circuitos elétricos do gêmeo digital.

Palavras-chave: Processamento Paralelo, Análise Nodal, CUDA.

ABSTRACT: This work presents a parallel processing approach to the solution of linear systems that represent electrical circuit nodes for the calculation of their voltages. Nodal analysis based on Kirchhoff's Node Law is used for modeling nodes in equations to form the linear system equivalent to each circuit. The linear system solution is achieved through parallel processing employing video card hardware and unified computing device architecture. The results obtained for the test 56,54%. It can be concluded that the parallel approach reduces the computational time for typical circuits by 56,54% compared to traditional processing.

Keywords: Parallel Processing, Nodal Analysis, CUDA.

1. Introdução

O desenvolvimento de um Gêmeo Digital (*Digital Twin*, DT), cópia virtual de processos reais [1], para o controle de operação de usinas hidrelétricas envolve a representação de circuitos elétricos em software. A análise de um circuito elétrico necessita da operação matricial com custo computacional da ordem de $O(n^3)$. A medida que o número de nós do circuito aumenta, o impacto no desempenho no DT é considerável, sendo necessário a busca de alternativas para a redução do tempo computacional para realizar as operações. O tempo computacional impacta no tempo do ciclo de simulação do DT, que deve ser o menor possível, proporcionando fidelidade na simulação, nos limites do hardware disponível. Um nó ou junção de um circuito é o ponto de ligação de dois ou mais elementos do circuito. Um nó une dois ou mais condutores.

Os métodos de análise de circuitos envolvem a lei de Ohm e conceitos matemáticos como a multiplicação e inversão matricial no contexto da resolução de sistemas lineares e a solução de equações com a descrição do comportamento do circuito a partir das variáveis de tensão, corrente e resistência [2]. O equacionamento de um circuito é baseado nas Leis de Kirchhoff que descrevem o comportamento das tensões nas malhas e das correntes nos nós do circuito. Uma malha é um caminho fechado de um circuito ou qualquer caminho fechado de um condutor [3]. Os conceitos de malha e nó dividem a Lei em de Kirchhoff em outras duas Leis conhecidas como Lei das malhas de Kirchhoff (*Kirchhoff Voltage Law* - KVL) e a Lei dos nós de Kirchhoff (*Kirchhoff Current Law*, KCL) [4].

A análise nodal é um procedimento geral para análise de circuitos a partir do uso de tensões dos nós como variáveis do circuito. Escolhido um nó qualquer do circuito de referência, ponto de potencial zero ou terra, os demais nós do circuito têm potência fixa em relação à referência. As interconexões tem resistência zero e todos os pontos ligados a um nó tem a mesma tensão elétrica [5].

A análise nodal é realizada nas etapas descritas a seguir. Encontrar o número de nós presentes no circuito. O número de equações necessárias para efetuar a análise do circuito. Para um circuito com *n* nós vão existir $(n-1)$nós com um potencial fixo em

relação ao nó de referência escolhido. Cada um dos $(n-1)$ nós têm uma equação associada para realizar a análise do circuito [6]. Um nó do circuito é escolhido como nó de referência e é atribuído a ele uma tensão nula ou terra. Um nó com muitos ramos é candidato a ser nó de referência. É feita a escolha de um sentido arbitrário para a corrente para cada elemento do circuito e é lhe atribuída a sua polaridade. Aplica-se a KCL a todos os nós do circuito, exceto ao nó de referência. Finalmente, as tensões dos nós são obtidas a partir da solução do sistema de equações resultantes do equacionamento de cada nó [7].

A análise de malhas ou método das correntes das malhas é baseada na Lei KVL. A aplicação da análise de malhas tem como pré-requisito a planaridade do circuito, caso contrário, não é possível usar a análise de malhas. O circuito planar é desenhado em um único plano sem que dois ramos se cruzem [8]. A anáĺise de malhas consiste nas etapas descritas a seguir. Verificar a planaridade do circuito, caso o circuito não seja planar não possível aplicar a análise de malhas. Escolher de modo arbitrário, o sentido das correntes da malha [9]. O número de correntes arbitrárias deve ser igual ao número de ramos mais o número de nós do circuito. O número de equações é igual ao número de correntes, ou seja, igual a número de malhas no circuito analisado. Uma corrente da malha deve percorrer todos os elementos do circuito, passando preferencialmente apenas uma vez em cada elemento. Identificar a polaridade da tensão a cada ramo do circuito. Finalmente, obter uma equação para cada malha percorrendo o circuito no mesmo sentido da corrente [10].

A arquitetura de dispositivo de computação unificada (*Compute Unified Device Architecture*, CUDA), desenvolvida pela NVIDIA, possibilita o uso de computação paralela através do uso de uma extensão para a linguagem de programação C com o apoio de uma unidade de processamento gráfico (*Graphics Processing Unit,* GPU) [11].

A arquitetura CUDA incluí um *pipeline* (segmentação de instruções) unificado para permitir que cada unidade lógica e aritmética (*Arithmetic Logic Unit*, ALU) do chip seja agrupada por um programa para realizar cálculos de uso geral [12]. O NVIDIA pretendia que a nova família de processadores gráficos fosse usada para computação de uso geral [13]. As ALUs foram construídas com os requisitos de aritmética de ponto

flutuante de precisão única projetadas para o uso de um conjunto de instruções personalizado para computação geral e não específico para gráficos com poder de processamento paralelo devido as motivações de seu desenvolvimento [14].

As unidades de execução na GPU ganharam acesso arbitrário de leitura e gravação à memória e acesso a um *cache* gerenciado por software conhecido como memória compartilhada [11]. Todos esses recursos da arquitetura CUDA foram adicionados a uma GPU para ganhar desempenho em cálculo e executar bem as tarefas gráficas tradicionais [15]. Quando comparado ao *pipeline* de um processador de dados de uma unidade central de processamento (*Central Processing Unit,* CPU) tradicional, a execução de cálculos de uso geral em uma GPU é um novo conceito [16].

As aplicações CUDA envolvem a solução de problemas em diferentes áreas como a dinâmica de fluidos, simulação de modelos climáticos, ramos da criptografia como o de cripto-moedas e traço de raios para estudos geofísicos [12].

O objetivo deste trabalho é apresentar uma alternativa à computação tradicional para a análise nodal empregando a tecnologia GPU CUDA. Pretende-se resolver os sistemas lineares que representam os nós dos circuitos elétricos em análise no contexto de um DT para o controle de operação de uma Usina Hidrelétrica ora em desenvolvimento.

Materiais e Métodos

O principal problema a ser superado neste trabalho é a solução dos sistemas de equações lineares para obter as tensões dos nós dos circuitos são processados pelo módulo do sistema eletrônico de potência do DT sendo desenvolvido para o controle de operação da Usina Hidrelétrica de Jirau instalada no Rio Madeira no Estado de Rondônia, com aplicação para treinamentos, pré-operação, dentre outras. Estima-se que os circuitos possam atingir e impactar o desempenho do sistema, visto que este módulo do sistema é crítico, ou seja, outros modelos dependem da resolução do circuito para seguir o seu fluxo. Vale ressaltar, que os sistemas lineares típicos do DT possuem coeficientes complexos e uma estratégia para sua resolução é dividi-los em dois

sistemas. Um sistema com a parte real e outro sistema com a parte complexa onde as variáveis são números complexos ou fasores.

Neste trabalho a análise de circuito será implementada usando tecnologia GPU CUDA para circuito da literatura como estudo de caso para que na sequência do estudo o software desenvolvido possa ser modificado e melhorado para a análise de circuitos reais no ambiente DT. A metodologia empregada para implementar o software experimental será a análise nodal porque a análise de malha só é utilizada para circuitos planos. O ambiente DT requer a análise de circuitos não planos.

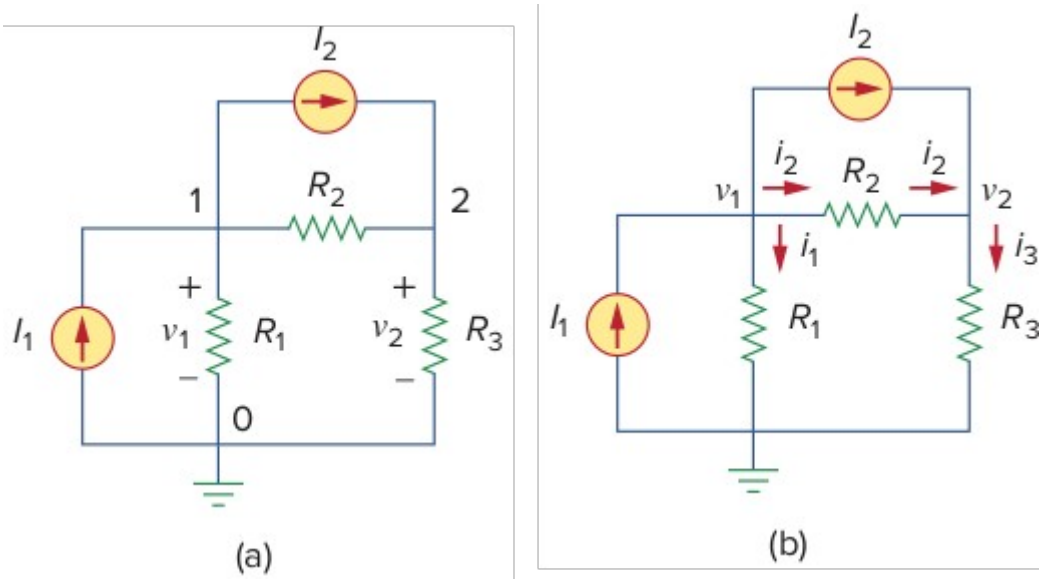

Figura 1. Circuito para análise nodal, Adaptado de Alexander [5].

O objetivo da análise nodal é encontrar as tensões dos nós. Assim, dado um circuito com n nós sem fontes de tensão, a análise nodal do circuito envolve as seguintes etapas. 1) Escolha um nó como o nó referência (terra). Atribua as tensões $v_1, v_2, \ldots, v_n$ para os $n-1$ restantes [17]. As tensões são referenciadas em relação ao nó de referência; 2) Aplique a KCL a cada um dos $n-1$ nós não referência. Use a Lei de Ohm para expressar as correntes da ramificação em termos das tensões dos nós; 3) Resolver as equações simultâneas resultantes do circuito para obter a tensões conhecidas dos nós [18].

O primeiro passo da análise nodal é selecionar um nó de referência. O nó de referência é chamado de terra. Assume-se que o nó de referência tem potencial zero. O aterramento do chassi é usado em dispositivos onde o gabinete ou chassi atua como um

ponto de referência para todos os circuitos. Depois de selecionar um nó de referência, é atribuída a tensão oas nós não referência.

No circuito da Figura 1(a), o nó 0 é o nó de referência ($v=0$). Os nós 1 e 2 recebem as tensões v_1e v_2, respectivamente. As tensões do nó são definidas em relação ao nó de referência [19]. A Figura 1(a), mostra cada tensão do nó é o aumento da tensão do nó de referência para o nó não referência correspondente ou simplesmente a altitude desse nó em relação ao nó de referência.

A segunda etapa da análise nodal é feita aplicação da KCL a cada nó não referência do circuito. A Figura 1(b) mostra que i_1, i_2 e i_3 não são somados como as correntes através dos resistores R_1, R_1e R_3,respectivamente. A aplicação da KCL para o nó 1 obtém-se

$$I_1 = I_2 + i_1 + i_2. \tag{1}$$

Até o nó 2, tem-se

$$I_2 + i_1 = i_3. \tag{2}$$

Nessa etapa é aplicada a lei de Ohm para expressar as correntes desconhecidas i_1, i_2e $i_3$em termos das tensões nos nós [20]. Como a resistência é um elemento passivo, pela convenção dos sinais passivos, a corrente deve fluir de um potencial mais alto para um potencial mais baixo. Esse princípio é expresso por

$$i = \frac{v_{alto} - v_{baixo}}{R}.$$

(3)

A partir desse princípio, a partir da Figura 1(b), obtém-se

$$i_1 = \frac{v_1 - 0}{R_1} \text{ ou } i_1 = G_1 v_1$$

$$i_2 = \frac{v_1 - v_2}{R_2} \text{ ou } i_2 = G_2(v_1 - v_2) \tag{4}$$

$$i_3 = \frac{v_2 - 0}{R_3} \text{ ou } i_3 = G_3 v_2$$

Substituindo a Equação (4) na Equação (1) e Equação (2), tem-se, respectivamente

$$I_1 = I_2 + \frac{v_1}{R_1} + \frac{v_1 - v_2}{R_2} \tag{5}$$

$$I_2 + \frac{v_1 - v_2}{R_2} = \frac{v_2}{R_3}. \tag{6}$$

Em relação às condutâncias, Equação (5) e (6), tem-se

$$I_1 = I_2 + G_1 v_1 + G_2 (v_1 - v_2) \tag{7}$$

$$I_2 + G_2 (v_1 - v_2) = G_3 v_2. \tag{8}$$

A terceiro etapa da análise nodal é a resolução das tensões dos nós. Aplica-se a KCL aos $n-1$nós não referência. Obtém-se $n-1$equações simultâneas, como Equações (5) e (6) ou (7) e (8) [21]. Para o circuito da Figura 1, a solução das Equações (5) e (6) ou (7) e (8) produz as tensões dos nós v_1e $v_2$usando qualquer método padrão, como o método de substituição, o método de eliminação, a regra de Cramer ou a inversão da matriz [22]. Para usar um dos dois últimos métodos, é necessário converter as equações simultâneas na forma de matricial. As Equações (7) e (8) podem ser escrita na forma matricial

$$\begin{bmatrix} G_1 + G_2 & -G_2 \\ G_2 & G_2 + G_3 \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \end{bmatrix} = \begin{bmatrix} I_1 - I_2 \\ I_2 \end{bmatrix} \tag{9}$$

Considerando o circuito da Figura 1(a) para o uso da análise nodal e ganho computacional, tem se a multiplicação de matrizes em paralelo ou seja resolvendo cada nó paralelamente. Onde o nó1 e nó2 do nosso exemplo tem sua resolução vinculada a um construtor único, o qual fica responsável pela resolução dos cálculos. Para a divisão do problema existem kernels responsáveis por executar cada função repassada pela sintaxe da API CUDA. A definição por usar núcleos da API CUDA é feita pela declaração do recurso do pacote *numba* do python. Isso significa que os dados são

transferidos para a memória da GPU paralelizando as tarefas de resolução dos nós através do acesso à memória global da GPU.

A solução do sistema linear da Equação (9) pode ser feita usando recursos GPU CUDA com processamento em paralelo através da distribuição das *threads* que são uma das características da arquitetura CUDA [23].

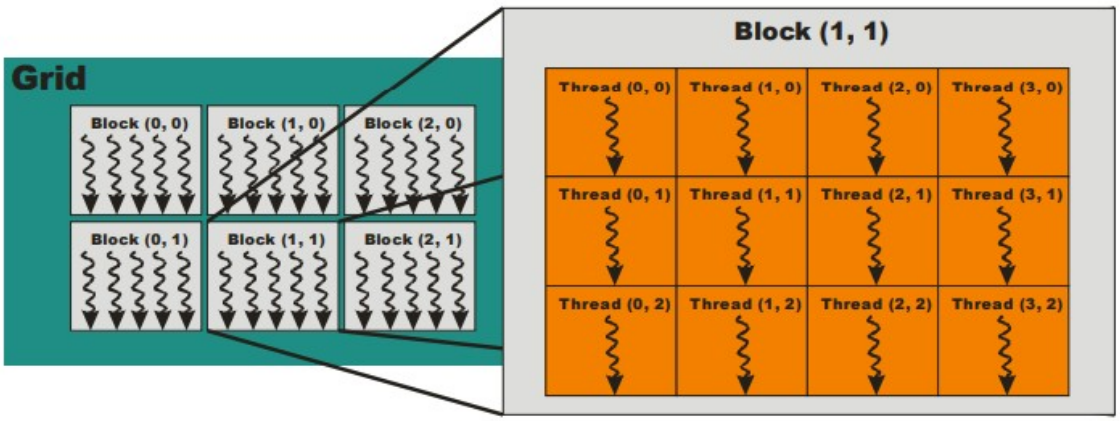

Figura 2. Elementos da arquitetura CUDA, Fonte NVIDIA CUDA [24].

A Figura 2 mostra os principais elementos da arquitetura CUDA. O processo de distribuição das *threads* em CUDA ocorre pela definição de *blocks* no *grid* da placa gráfica. O *grid* é a configuração da placa para a execução de um *kernel*. *Block* forma o conjunto de *threads* que serão processadas simultaneamente em memória compartilhada. O processo de comunicação entre *blocks* ocorre através da memória global, a qual possui desempenho inferior a memória compartilhada. a identificação de *blocks* e *threads* é composto de identificadores tridimensionais, através destes o desenvolvedor tem acesso a *thread* a que deve processar [25].

No processo adotado na programação em python foi feita a implementação do Algoritmo 1 (abaixo) que é responsável pela multiplicação de matrizes, necessário para a resolução de sistema linear implicando em se computar a inversão da matriz, sendo esta uma matriz quadrada por resolução pela análise nodal.

Algoritmo 1 - Multiplicar matriz

```
1       Entrada:
2           A = matriz que está sendo multiplicada
3           B = matriz com os valores de potência
4       Saída:
5           C = matriz com os resultados dos cálculos
6           R = matriz de resultado
7       # função para multiplicar uma matriz por um vetor
8       # função decorada para cuda
9       Função  = multiplicar vetor( A, b ):
10          C = A * b
11          Retornar C
12      # função para multiplicar duas matrizes
13      Função = multiplicar matriz( A, B ):
14          # Para  nas colunas da matriz 'B'
            Para inicio até fim colunas(b) faça
            R[:, i] = multicar vetor( A, B[:,i] )
            Retornar R
```

No início do algoritmo temos os parâmetros de entrada compostos pela matriz **A** e **B**, o resultado delas é uma saída **R,** que corresponde a multiplicação da matriz **A** pelo inversa da matriz **B**. Na linha 2 pode-se informar os valores que compõem a matriz **A**, na linha 3 temos os valores da matriz **B**. Na linha 11 temos a função que processa a

multiplicação direto na arquitetura CUDA, através da decoração da função executando o processamento na GPU e devolvendo para a CPU e retornando o resultado no vetor **C**. A linha 16 é responsável por invocar o processamento feito na linha 11 para cada elemento da matriz a serem calculados. Foram criadas funções que são passadas para a API CUDA para serem processadas as matrizes na GPU e enviadas ao CPU, para resolver os circuitos elétricos pela análise nodal na programação paralela.

2. Resultados e Discussão

Foi implementado algoritmo utilizando a linguagem Python, onde utilizou-se inicialmente apenas o processamento de CPU, posteriormente foi implementado o algoritmo utilizando a GPU, através da biblioteca *Numba* (utiliza API CUDA) em Python, estes códigos foram disponibilizados no github (https://github.com/bierley123/pythonapicuda/blob/master/PythonCuda.py).

A tabela 1 mostra o cada tamanho de cada matriz utilizada e seus respectivos tempos. As mesmas matrizes são processadas tanto em programação tradicional usando CPU quanto programação paralela usando GPU (utiliza API CUDA).

Tabela 1: Tempo processamento CPU x GPU em segundos.

Tamanho Matriz	Linguagem Python tradicional	Linguagem Python com Gpu	Diferença Tradicional - Paralelo	Percentual %
100 x 100	0.1274585723876953	0.02337193489074707	0.10408663749694824	44,35
500 x 500	4.8812413215637210	1.18661522865295400	3.69462609291076660	311,36
1000 x 1000	23.7649521827697750	7.53870677947998050	16.22624540328979500	215,24
1500 x 1500	59.0981595516204800	23.89720773696899400	35.20095181465149000	147,30
2000 x 2000	120.9714651107788100	52.77848839759827000	68.19297671318054000	129,20
2500 x 2500	182.7194447517395000	97.07695484161377000	85.64248991012573000	88,22
3000 x 3000	288.8846502304077000	176.87205576896667000	112.01259446144104000	63,33
3500 x 3500	441.4162542819977000	266.10677719116210000	175.30947709083557000	65,88
4000 x 4000	619.8696622848511000	408.52408576011660000	211.34557652473450000	51,73
4500 x 4500	831.1901185512543000	560.63767409324650000	270.55244445800780000	48,26
5000 x 5000	1111.0693798065186000	761.98827505111700000	349.08110475540160000	45,81
5500 x 5500	1411.9509959220886000	908.88023877143860000	503.07075715065000000	55,35
6000 x 6000	1809.7701418399810000	1426.73860120077332000	383.03154063224790000	26,85
6500 x 6500	2146.5558035373690000	1642.31782841682430000	504.23797512054443000	30,70
7000 x 7000	2559.5322494506836000	1789.08699560165400000	770.44525384902950000	43,06
7500 x 7500	3040.4100060462950000	2208.05205607414250000	832.35794997215270000	37,70
8000 x 8000	6632.7703993320465000	3265.03331494331000004	3367.73708438873250000	103,14
Média	**1252.0577872220208000**	**799.8081910469953**		**56,54**

No gráfico 1 temos a curva do custo computacional, com o respectivo comportamento e speedup de cada uso da linguagem tradicional e paralela.

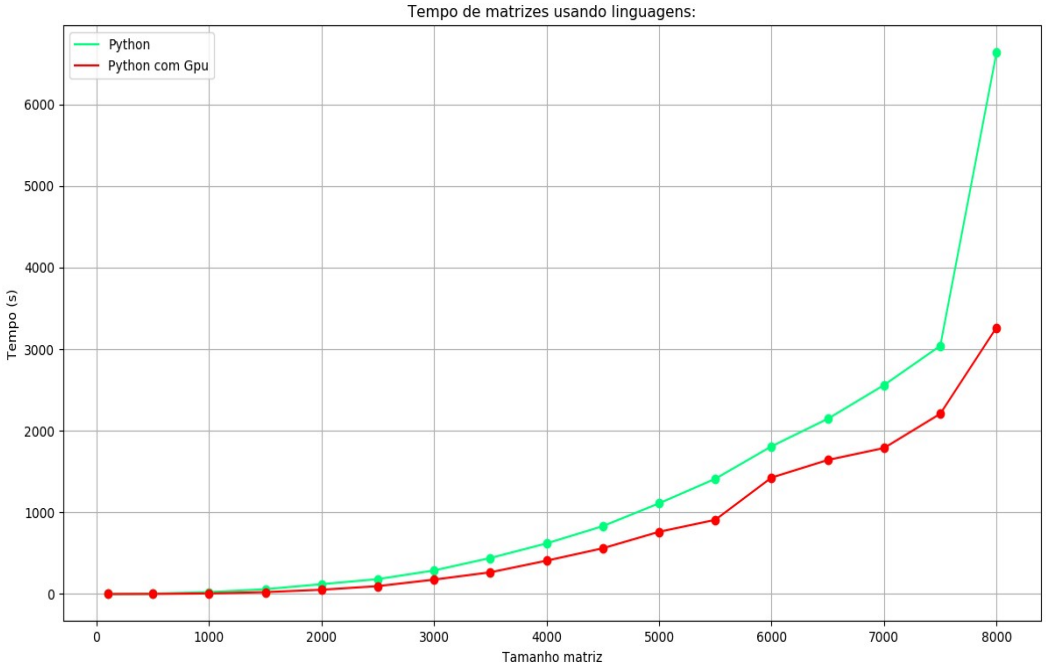

Figura 3. Custo computacional do algoritmo.

Utilizando as funções CUDA para realizar as operações na geração das matrizes, o tempo médio do cálculo destas matrizes foi reduzido 56,54% vezes mais rápido. Os resultados mostraram que mesmo realizando uma pequena parte do processamento na GPU a combinação da linguagem interpretada com o processamento na GPU oferece uma alternativa bastante conveniente para o desenvolvimento de algoritmos numéricos.

3. Conclusão

O resultado obtido mostra que o processamento paralelo minimiza o impacto computacional para a resolução de sistemas lineares com números reais e complexos a partir da implementação da análise nodal em paralelo empregando a tecnologia CUDA. Os métodos de análise nodal permitem a utilização de programação em *threads* que possibilitam a independência na sua resolução, permitindo que possam ser processados como tarefas paralelizadas, sendo que o seu desempenho em GPUs, apresentam um menor custo computacional em comparação ao processamento não paralelizado, visto que o processamento paralelo tende a se estabilizar mesmo com o crescimento das operações executadas.

O poder computacional paralelizado demonstra que é possível que as análises sejam mais esparsas e numerosas, isto é devido ao menor tempo de processamento proporcionado pela arquitetura CUDA, assim aumentando a possibilidade de soluções com maior confiabilidade. Devido a grande quantidade de informações armazenadas nas matrizes que representam circuitos elétricos faz se necessário o uso de técnicas de computação de alto desempenho e de computação paralela. Com o uso da arquitetura de computação paralela CUDA, obteve-se um desempenho próximo ao de supercomputadores em um computador pessoal. É possível concluir que a abordagem paralela reduz o tempo computacional para circuitos típicos em comparação com processamento não paralelizado.

4. Agradecimentos

Os autores agradecem à Energia Sustentável do Brasil pelo apoio para realização desse estudo sob contrato ANEEL PD-06631-0007/2018 e contrato Jirau 064/2018.

5. Referências

1. Kritzinger, W., Karner, M., Taar, G. Henjes, J., Sihn, W. Digital Twin in Manufacturing: A categorical literature review and classification, IFAC-PapersOnLine, v. 51, n. 11, p. 1016-1022, 2018.

2. Irwin, D., Nelms, R. M. Basic Engineering Circuit Analysis, Wiley, 2015.

3. O'Malley, J. R., Basic Circuit Analysis, McGraw-Hill, 2011.

4. Boylestad, R. Introductory Circuit Analysis, Pearson, 2015.

5. Alexander, C. K., Sadiku, M. N. O., Fundamentals of Electric Circuits, McGraw-Hill, 2017.

6. Alba-Flores, R. Circuit Analysis Lab,Amazon Digital Services LLC, 2013.

7. Simões, M. G., Farret, F. Modeling Power Electronics and Interfacing Energy Conversion Systems, Willey, 2016.

8. Salam, A., Rhaman, Q. M. Fundamentals of Electrical Circuit Analysis, Springer, 2018.

9. Kanoussis, D. P., Analysis of Electric Circuits Amazon Digital Services LLC, 2017.

10. Vlach, J., Linear Circuit Theory: Matrices in Computer Applications, Apple Academic Press, 2016.

11. CUDA Compute Unified Device Architecture Programming Guide, NVIDIA,version 2.3. [Online]. Available: http://developer.download.nvidia.com/compute/cuda/2 3/toolkit/ docs/NVIDIA CUDA Programming Guide 2.3.pdf, 2009.

12. Mivule, K., Harvey, B., Cobb, C., Sayed, H. E., A Review of CUDA, MapReduce, and Pthreads Parallel Computing, Models, arXiv:1410.4453.

13. Bakhoda, A., Yuan, G. L., Fung, W. W. L., Wong, H., & Aamodt, T. M. Analyzing CUDA workloads using a detailed GPU simulator. 2009 IEEE International Symposium on Performance Analysis of Systems and Software. doi:10.1109/ispass.2009.4919648, 2009.

14. Lindholm, E., Nickolls, J., Oberman, S., & Montrym, J. (2008). NVIDIA Tesla: A Unified Graphics and Computing Architecture. IEEE Micro, v. 28, n 2, p. 39–55. doi:10.1109/mm.2008.31.

15. Ruetsch, G., Micikevicius, P., Optimizing Matrix Transpose in CUDA, NVIDIA CUDA SDK Application Note, 200, 2009.

16. August, D. I., Malik, S., Peh, L.-S., Pai, V., Vachharajani, M., Willmann, P., Achieving structural and composable modeling of complex systems, International Journal of Parallel Programming, v. 33, n. 2, p. 81–101, 2005. [Online]. Available: http: //dx.doi.org/10.1007/s10766-005-3569-3.

17. Smith, D. Circuit Analysis for Complete Idiots, Beyond Whyy , 2019.

18. Robbins, A. H., Mille, W. C., Circuit Analysis: Theory and Practice, Cengage Learning, 2012.

19. Thomas, R. E., Rosa, A. j., Toussaint, G. J., The Analysis and Design of Linear Circuits, Wiley, 2011.

20. Hayt, W., Kemmerly, J. Engineering Circuit Analysis, McGraw Hill 2018.

21. Nahvi, M., Edminister, J., Schaum's Outline of Electric Circuits, seventh edition, McGraw-Hill, 2017.

22. Zill, D. G., Advanced Engineering Mathematics, Jones & Bartlett Learning, 2016.

23. Hecquet, N., Parallel Calculus in CUDA: Heat equation, Black and Scholes Model, Éditions universitaires européennes, 2019.

24. NVIDIA CUDA C ProgrammingGuide. NVIDIA Corporation, ed. 5.0. [S.l.]: NVIDIA Corporation, 2013.

25. Sanders, J., Kandrot, E, CUDA by Example a Introdution to a General-Porpose GPU Programming, Addison Wesley, 2010.

Anexos

Exemplo 1:

```
import numpy as np
from timeit import default_timer as timer

def pow(a, b, c):
   for i in range(a.size):
       c[i] = a[i] ** b[i]

def main():
   vec_size = 100000000
   a = b = np.array(np.random.sample(vec_size), dtype=np.float32)
   c = np.zeros(vec_size, dtype=np.float32)
   start = timer()
   pow(a, b, c)
   duration = timer() - start
   print(duration)

if __name__ == '__main__':
   main()
```

Exemplo 2:

```
    import numpy as np

    # from numba import vectorize

    from numba import jit

    from parallelo import DimensaoMatrixError

    @jit(nopython=True)

    def mul_cuda(obj, other):

       return obj * other

    @jit(nopython=True)

    def soma_cuda(obj):

       return np.sum(obj)

    class MatrixMulCuda(np.ndarray):

       def __new__(cls, array):

          obj = array.view(cls)

          return obj

       def __array_finalize__(self, obj):
```

```
    if obj is None: return
    if not isinstance(obj, np.ndarray):
        raise "A matrix deve ser do tipo array: np.ndarray"
    self.obj = obj
    self.info = getattr(obj, 'info', None)
def __str__(self):
    texto = np.array2string(self.obj, separator=',')
    return ''.join(texto.splitlines())


def __mul__(self, other):
    if isinstance(other, (np.ndarray, self.__class__)):
        return self.mul_matrix(other)
    elif isinstance(other, (np.int, np.int8, np.int16, np.int32, np.int_)):
        return self.mul_scalar(other)
    else:
        raise "A operação deve ser relizada por uma matriz ou um vetor"
def mul_scalar(self, other):
    rows, cols = self.obj.shape
    C = np.zeros((rows, cols))
    for j in range(cols):
        col = self.obj[:,[j]]
        C[:,[j]] = col * other
    return C
def mul_matrix(self, other):
    rows, k, m = self.obj.shape[0], self.obj.shape[1], other.shape[1]
    if not k == other.shape[0]:
        raise DimensaoMatrixError(self.obj.shape, other.shape)
    C = np.zeros((rows, m))
    for ii in range(rows):
```

```
    for jj in range(m):
        aa = self.obj[ii,:]
        bb = other[:,jj]
        C[ii,jj] = soma_cuda(mul_cuda(aa,bb))
return C
```