**Pontifícia Universidade Católica de Goiás**

**Programa de Pós-Graduação Stricto Sensu em Engenharia de Produção e Sistemas**

Análise Nodal de Circuito Empregando GPU CUDA em Sistema Gêmeo Digital para Usina Hidrelétrica.

Bierley Souza Machado

Junho de 2020

Análise Nodal de Circuito Empregando GPU CUDA em Sistema Gêmeo Digital para Usina Hidrelétrica.

Bierley Souza Machado

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Engenharia de Produção e Sistemas da Pontifícia Universidade Católica de Goiás, como parte dos requisitos para obtenção do título de Mestre em Engenharia de Produção e Sistemas.

Orientador: Clarimar José Coelho, Dr.

Coorientador: Arlindo Rodrigues Galvão Filho, Dr.

Goiânia

Junho de 2020

Análise Nodal de Circuito Empregando GPU CUDA em Sistema Gêmeo Digital para Usina Hidrelétrica.

.

Bierley Souza Machado

Esta Dissertação julgada adequada para obtenção do título de Mestre em Engenharia de Produção e Sistemas, e em sua forma final pelo Programa de Pós-Graduação em Engenharia de Produção e Sistemas da Pontifícia Universidade Católica de Goiás em 20 de Junho de 2020.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Prof. Marcos Lajóvic Carneiro, Dr.

Coordenador do Programa de Pós-Graduação em Engenharia de Produção e Sistemas

Banca Examinadora:

Prof. Clarimar José Coelho, Dr. Orientador

Prof. Rafael Viana Carvalho Dr. Avaliador Externo

Prof.ª Solange da Silva. Dr.ª Avaliadora Interna - PUC Goiás

GOIÂNIA - GO

Junho de 2020

Machado, Bierley Souza

Análise Nodal de Circuito Empregando GPU CUDA em Sistema Gêmeo Digital para Usina Hidrelétrica – Goiânia – Goiás: PUC Goiás/MEPROS, 2020.

XXX p.: il.; 29,7 cm.

Orientador: Clarimar José Coelho

Dissertação (mestrado) – Pontifícia Universidade Católica de Goiás Programa de Mestrado em Engenharia de Produção e Sistemas, 2020.

Referências Bibliográficas: p.71-76.

Programa de Mestrado em Engenharia de Produção e Sistemas. Análise Nodal de Circuito Empregando GPU CUDA em Sistema Gêmeo Digital para Usina Hidrelétrica.

A Deus, que nos capacita até para os sonhos que parecem impossíveis; à minha esposa, Elaine Rodrigues Nogueira, a minha mãe Solange Nilce de Souza e tia Seleida Souza, que sempre me apoiaram, acreditaram no meu esforço e abriram mão de passar um tempo comigo para a concretização deste trabalho.

**Agradecimentos**

Ao meu orientador, Prof. Dr. **Clarimar José Coelho**, que me acompanhou em todo tempo da minha jornada.

Aos meus colegas e companheiros do MEPROS que sempre me incentivaram.

Ao Programa de Mestrado em Engenharia de Produção e Sistemas que me possibilitou realizar este trabalho.

A colaboração nesse projeto do Laboratório de Computação Científica da ECEC-PUC Goiás e a Energia Sustentável do Brasil S.A. (ESBR) e como recurso humano nesse projeto a coorientação do Professor Dr. **Arlindo Rodrigues Galvão Filho**.

À Fundação Aroeira e Usina Jirau, pelo estímulo a pesquisa e pelo financiamento do Projeto de Pesquisa & Desenvolvimento **PD-06631- 0007/2018** ao qual este projeto faz parte, intitulado como **”Desenvolvimento Experimental de Gêmeo Digital para Usinas Hidrelétricas”**.

**Resumo**

Em sistemas elétricos de potência é de grande valia ter-se um mecanismo que simula a operação deste sistema. Isto é realizado pela abordagem do Gêmeo Digital que para este trabalho, é uma representação virtual do sistema elétrico em tempo real. Permite-se assim, simular problemas e estratégias de operação para obter-se maior confiabilidade das ações e supervisão de equipamentos e da rede elétrica. Este trabalho contribui para a solução das equações de rede elétrica através de processamento computacional paralelo. Na abordagem de processamento paralelo para a solução de sistemas lineares, utiliza-se a análise nodal (Lei dos Nós de Kirchhoff). Isto permite obter com ganho computacional as tensões nos “Nós” e demais grandezas elétricas como a corrente e a potência. Sendo de extrema importância devido à operação em tempo real necessitar de alta disponibilidade, o que pode ser um problema em métodos convencionais, que podem apresentar baixa eficiência em comparação ao método de paralelismo que tendem a ser mais eficientes. 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 (CUDA). Porém, para esta modelagem necessitamos de um esforço computacional que pode inviabilizar seu uso no contexto de Gêmeo Digital. Neste contexto, a proposta de uso da programação paralela CUDApara a redução do tempo computacional gastos nos modelos. Com o uso dos modelos paralelizados, obtém-se uma elevação na performance temporal do Gêmeo Digital, permitindo monitorar e controlar objetos em tempo real, detecção de ataques cibernéticos, otimização de controles, validação de modelos, manutenção e medidas antecipadas e/ou contrárias. Equiparando o tempo de resposta ao da planta real. Os resultados obtidos serão incorporados em um modelo de Gêmeo Digital como previsto no projeto de Pesquisa Desenvolvimento Experimental de Gêmeo Digital da UHE-Jirau em Escopo Completo com Tecnologia de Processamento de Eventos Complexos Distribuídos para Investigações Sistêmicas (PD-06631-0007/2018).

**Palavras-chave:** Processamento paralelo, multiplicação de matrizes, sistemas lineares, ladrilhamento.

**Abstract**

In power electrical systems it is of great value to have a mechanism that simulates the operation of this system. This is accomplished by the Digital Twin approach which, for this work, is a virtual representation of the electrical system in real time. It is possible to simulate problems and operation strategies to obtain greater reliability of actions and supervision of equipment and the electrical network. This work contributes to the solution of electrical network equations through parallel computer processing. In the parallel processing approach for the solution of linear systems the nodal analysis is used (Kirchhoff's Node Law). This allows to obtain with computational gain the voltages in the nodes and other electrical quantities such as current and power. Being of extreme importance due to the operation in real time needs high availability, what can be a problem in conventional methods, that can present low efficiency in comparison to the parallelism method that tends to be more efficient. The solution of this linear system is obtained with parallel processing using the video card hardware and the unified computing device architecture (CUDA). However, for this modeling we need a computational effort that may make its use in the context of Digital Gemini unfeasible. In this context, the proposal to use CUDA parallel programming to reduce the computational time spent on the models. With the use of parallel models, we obtain an increase in the temporal performance of the Digital Gemini, allowing monitoring and control of objects in real time, detection of cyber attacks, optimization of controls, validation of models, maintenance and anticipated and or contrary measures. Equipping the response time to the real plant. The results obtained will be incorporated into a Digital Twin model as foreseen in the Digital Twin Experimental Research and Development project of UHE-Jirau in Full Scope with Distributed Complex Event Processing Technology for Systemic Investigations (PD-06631-0007/2018).

**Keywords:** Parallel processing, matrix multiplication, linear systems, tiling.

**Índice de Figuras**

Figura 1 - Circuito para análise nodal ................................................................................................................... 19

Figura 2 - SpeedUp................................................................................................................................................ 24

Figura 3 - Speedup proporção paralelizável........................................................................................................... 24

Figura 4 - Speedup calculado................................................................................................................................. 24

Figura 5 - Argumento de Amdahl ......................................................................................................................... 25

Figura 6 - Tempo de execução e speedup ............................................................................................................. 25

Figura 7 - Demonstração de multiplicação de matrizes ........................................................................................ 27

Figura 8 - Sistema paralelo .................................................................................................................................... 28

Figura 9 - Memória compartilhada ........................................................................................................................ 29

Figura 10 - GPU exposta em uma placa de vídeo ................................................................................................. 32

Figura 11 - As placas de vídeo têm milhares de núcleos para processar cargas em paralelo ............................... 33

Figura 12 - Divisão de tarefas de um problema entre as CPUs ............................................................................. 35

Figura 13 - Aceleração de código utilizando GPU ............................................................................................... 36

Figura 14 - Elementos da arquitetura CUDA ........................................................................................................ 36

Figura 15 - Hierarquia de funcionamento de memória em GPU .......................................................................... 37

Figura 16 - Demonstração da multiplicação de matrizes por ladrilhamento ......................................................... 41

Figura 17 - Planta digital de funcionando da Usina Hidrelétrica de Jirau ............................................................ 43

Figura 18 – Lado direito da Usina Hidrelétrica de Jirau ....................................................................................... 44

Figura 19 - Lado esquerdo da Usina Hidrelétrica de Jirau Jirau ........................................................................... 44

Figura 20 - Resolução de Sistemas de equações lineares com OpenMP .............................................................. 62

Figura 21 - Resolução de Sistemas de equações lineares com CUDA ................................................................. 63

Figura 22 - Comparativo OpenMP x CUDA ........................................................................................................ 64

Figura 23 - Custo computacional do algoritmo ..................................................................................................... 65

Figura 24 – Comparativo algoritmos de CPU e GPU A ....................................................................................... 66

Figura 25 – Custo computacional dos algoritmos na GPU A, B e C .................................................................... 67

Figura 26 – Comparativo dos algoritmos na GPU A, B e C ................................................................................. 67

**Índice de tabelas**

Tabela 1: Comparação da abordagem com eliminação Gaussiana x métodos clássicos.........................................10

Tabela 2: Comparação dos métodos diretos e iterativos em relação a vários aspectos...........................................21

Tabela 3: Valores para demonstrar a quantificação do desempenho computacional..............................................23

Tabela 4: Tempo processamento CPU x GPU em milissegundos..........................................................................65

Tabela 5: Tempo processamento diferentes GPUs em milissegundos....................................................................68

Tabela 6: Parâmetros de entrada do cuSolverDN...................................................................................................87

Tabela 7: Status retornado.......................................................................................................................................87

Tabela 8: Parâmetros de entrada do cuBlas .......................................................................................................... 88

Tabela 9: Status retornado.......................................................................................................................................88

Tabela 10: Parâmetros de entrada do cublasZgemm3m..........................................................................................88

Tabela 11: Parâmetros de entrada do cuSolverDnSgetrf e cuSolverDnSgetrs........................................................90

Tabela 12: Parâmetros de entrada para cuSolverDnDgeqrf e cuSolverDnDorgqr..................................................91

Tabela 13: Parâmetros de entrada do cuSolverDnSpotrf e cuSolverDnSpotrs.......................................................92

Tabela 14: Parâmetros de entrada do cusolverDnSsyevj e cusolverDnDsyevj.......................................................92

Tabela 15: Parâmetros de entrada do cuSparce.......................................................................................................94

Tabela 16: Parâmetros de entrada do cuSolverDnSsyevd.......................................................................................95

**Lista de siglas**

ALU Unidade de Lógica Aritmética

AFIPS Conferência conjunta de informática unificada

API Interface de programação de aplicação

BLOCK Conjunto de threads que serão processadas simultaneamente

CACHE Arquivo temporário

CPS Sistema ciber físico

CUDA Arquitetura unificada de dispositivos

CPU Unidade Central de Processamento

CLOCK Referencial de potência

DIGITAL TWIN Gêmeo digital

DT Gêmeo digital

DRAM Memória dinâmica de acesso principal

GRID Configuração da placa para a execução de um kernel

GPU Unidade Processadora Gráfica

HYPERTHREADING Tecnologia proprietária da Intel usada para computação paralela em processadores x86.

INTRANET Uma rede de computadores privada

INTERNET Sistema global de redes de computadores interligadas

KCL Lei de Kirchhoff

KERNEL Núcleo do sistema operacional

KVL Lei de Tensão Kirchhoff

MATLAB Software interativo voltado para o cálculo numérico

PIPELINE Busca de uma ou mais instruções além da próxima a ser executada

OPENACC Aceleradores abertos uma forma de programação para computação paralela

OPENCL Arquitetura para escrever programas que funcionam em plataformas heterogêneas

OPENMP Multi-Processamento Aberto

SM Fluxo Multiprocessadores

SPEEDUP Aumento da velocidade

SPICE Programa de simulação com ênfase em circuito integrado

THREADS Fluxos de processamento independentes

THROUGHPUT Taxa de transferência

UHE Usina hidrelétrica

Sumário

[Capítulo 1 – Introdução 1](#_Toc46140275)

[Capítulo 2 – Sistemas de equações lineares 9](#_Toc46140276)

[2.1 – Análise Nodal 9](#_Toc46140277)

[2.2 – Algoritmos solucionadores de equações lineares 12](#_Toc46140278)

[2.3 – Circuito para análise nodal 18](#_Toc46140279)

[2.4 – Resolução de circuitos pela análise nodal modificada 19](#_Toc46140280)

[2.5 – Considerações do Capítulo 22](#_Toc46140281)

[Capítulo 3 – Programação Paralela 23](#_Toc46140282)

[3.1 – Capacidade de processamento 23](#_Toc46140283)

[3.2 – Multiplicação de matrizes 27](#_Toc46140284)

[3.3 – Abordagem de programação paralela 27](#_Toc46140285)

[3.4 – Solução do sistema linear em GPU CUDA 36](#_Toc46140286)

[3.5 – Gêmeo Digital 41](#_Toc46140287)

[3.6 – Considerações do Capítulo 45](#_Toc46140288)

[Capítulo 4 – Materiais e Métodos 46](#_Toc46140289)

[4.1 – Método cientifico adotado 46](#_Toc46140290)

[4.2 – O processo de investigação com técnicas, bibliotecas e ferramentas 48](#_Toc46140291)

[4.3 – Experimentos 49](#_Toc46140292)

[4.4 – O processamento dos dados 50](#_Toc46140293)

[4.5 – Análise e interpretação dos resultados através dos algoritmos 53](#_Toc46140294)

[4.6 – Considerações do Capítulo 61](#_Toc46140295)

[Capítulo 5 – Resultados 62](#_Toc46140296)

[5.1 – Avaliar e analisar métodos diretos e iterativos no CPU 62](#_Toc46140297)

[5.2 – Avaliar e analisar métodos diretos e iterativos na GPU 63](#_Toc46140298)

[5.3 – Comparar os experimentos executados na CPU x GPU 64](#_Toc46140299)

[5.4 – Comparar a paralelização em diferentes GPUs 66](#_Toc46140300)

[5.5 – Considerações do Capítulo 68](#_Toc46140301)

[Capítulo 6 – Conclusões e trabalhos futuros 69](#_Toc46140302)

[Referências Bibliográficas 71](#_Toc46140303)

[Apêndice A - Solucionadores de equações lineares 77](#_Toc46140304)

[Apêndice B – Bibliotecas 87](#_Toc46140305)

[Apêndice C – Algoritmo proposto com ladrilhamento 96](#_Toc46140306)

# Capítulo 1 – Introdução

A análise de circuito ou solução de um circuito é a descoberta das tensões e correntes de cada elemento do circuito. O termo circuito tem origem na palavra círculo*.* Um circuito se refere a uma porção de componentes reais, fontes de potência e fontes de sinal conectados para que a corrente flua em um círculo completo. Um circuito elétrico é feito de elementos*.* Elementos podem ser fontes, resistor, capacitor e incluem pelo menos uma fonte*.* A fonte é conectada a vários componentes. As fontes fornecem energia para um circuito. São dois os tipos de fontes, de tensão e de corrente (SADIKU, 2017). Depois de simplificar um circuito o máximo possível, todos os métodos de análise são uma versão da estratégia. Criar um conjunto de equações independentes, com base nos elementos e conexões do circuito, resolver o sistema de equações simultâneas para as variáveis independentes (tensões ou correntes) usando álgebra linear. Resolver as tensões e correntes individuais restantes (SADIKU, 2020). Os métodos para a análise de circuitos são aplicação direta das leis de Ohm e Kirchoff, método das tensões de nó, métodos das correntes de malha simples e método das correntes de malha. A aplicação direta das leis fundamentais é rápido e funciona para circuitos simples. É pouco eficiente quando o circuito é mais complexo. Os métodos das tensões no nó e o método da corrente na malha simples minimizam o número de equações simultâneas e são eficientes para circuitos complexos (NAGEL, 1971).

A análise nodal de circuito ou tensão de análise de um nó ou método do ramo atual é um método para determinar a tensão ou [diferença de potencial](https://pt.wikipedia.org/wiki/Tensão_elétrica) entre nós, pontos onde os elementos ou ramos se ligam, em um [circuito elétrico](https://pt.wikipedia.org/wiki/Circuito_elétrico). A análise nodal modificada (*Modified Nodal Analysis*, MNA) é uma extensão da [análise nodal](https://pt.qwe.wiki/wiki/Nodal_analysis) (*Nodal Analysis*, NA) que não só determina tensões no nó do circuito, mas também algumas correntes de ramo. A MNA foi desenvolvida como um formalismo para mitigar a dificuldade de representar os componentes definidos tensão em análise nodal (por exemplo, tensão controlado por fontes de tensão). Outros formalismos, como quadro esparso, são igualmente gerais e relacionados através de transformações da matriz (HO, 1975).

A NA é uma técnica para formular equações de circuito, um método clássico que produz uma diagonal numericamente bem comportada, ou seja, os valores dos números são relativamente próximos. É portanto, muito popular e tem sido amplamente utilizado em programas de computador como visto em Nagel (NAGEL, 1971) e Calhan (CALAHAN, 1972). Porém, existe uma limitação em sua forma básica, a forma como trata as fontes de tensão de maneira ineficiente, sendo incapaz de incluir elementos dependentes da corrente linear ou não linear.

A MNA técnica amplamente utilizada para modelagem de circuitos. Foi introduzido pela primeira vez em Ho et al (HO, 1975). É baseada em considerar um circuito como um gráfico e resulta em um modelo algébrico diferencial. Este método fornece uma estrutura que permite uma análise matematicamente elegante das propriedades essenciais e sua interpretação física. (HO, 1975).

Utilizando uma linguagem de programação para resolver a MNA ao invés de NA, foi observado que o método de segmentação poderia modelar toda uma estrutura para análise dos circuitos. Podendo assim, ter sua velocidade de cálculo aumentada aplicando o método de MNA. Com isso a operação de todo o processo em programas de computador e em termos de precisão ter a capacidade de utilizar o método de MNA para encontrar os fatores comuns e removê-los. Assim, todas as divisões com restos, são eliminadas categoricamente permitindo remover possíveis erros de arredondamento e contribuindo para o aumento da precisão com esta técnica (LIN, 2019).

Várias tentativas foram feitas no passado para generalizar o método NA. Nagel (NAGEL, 1971) tratou cada uma das tensões como independentes e as substituiu por fontes equivalentes em todos os ramos conectados ao nó positivo. Calhan (CALHAN, 1972) utilizou geradores para converter indutores lineares e não lineares para capacitores no domínio do tempo. Além disso, resistências extremamente pequenas ou negativas foram introduzidas em alguns programas para acomodar as atuais dependências. Além dos erros de arredondamento, outra desvantagem da NA é que as correntes de ramificação não são precisas o suficiente para serem usadas em programas de computador. Estas complicações podem ter contribuído para o fato de que outros métodos fossem desenvolvidos como a MNA, gerando uma demanda computacional (HO, 1975).

A demanda computacional tem sido intensificada devido as evoluções tecnológicas. Sendo necessárias evoluções tecnológicas nas áreas de *hardware*, microeletrônica e computação em geral, onde o conceito de sistemas ciberfísicos (*Cyber Physcial Systems,* CPS) tem ganhando importância. O fundamental deste conceito e a possibilidade de trabalho conjunto entre sistemas computacionais distribuídos e processos físicos, contribuindo para a revolução industrial pautada na descentralização e paralelização de processos e de dispositivos (LEE, 2008).

Devido a necessidade de novos métodos de produção, miniaturização de eletrônicos, novos sensores de tecnologias e a internet das coisas (*Internet of Things*, IoT) que contribuíram para o desenvolvimento de novos equipamentos. Como consequência tem-se produtos cada vez mais complexos e também oportunidades únicas. Por outro lado, exigem mais em termos de eficiência para sistemas complexos. O surgimento de novas tendências tecnológicas, como a inteligência artificial (*Artificial Intelligence*, AI), IoT, computação nas nuvens e experiências imersivas (realidade aumentada), permitiram que muitos dos paradigmas fossem quebrados. Junto a essas novas tendências tecnológicas, também estão os gêmeos digitais (HARTMANN, 2020).

Dentre as evoluções tecnológicas impulsionadas pelo CPS, destaca-se o conceito de gêmeo digital (*Digital Twin*, DT), cópia virtual de processos reais (KRITZINGER, 2018), onde uma de suas características e a de ser uma representação virtual de objetos físicos (KRITZINGER, 2018). Assim existe uma relação direta de paralelismo dos processos e com os conceitos da Indústria 4.0 (KAGERMANN, 2013).

Para reduzir os esforços que sejam necessários para processar inúmeras tarefas, é possível combinar algumas técnicas. Como uma técnica de integração eficiente, baseada em uma estratégia aplicada, designada de método voxel (YANG, 2012), que permite que códigos tenham acessos eficientes à memória, possibilitando paralelização e transferência rápida para novas arquiteturas de computação como a unidade de processador gráfico (*Graphics Processor Unit*, GPU). Solucionadores geométricos de grades múltiplas que possibilitam soluções ótimas para sistemas lineares em termos de complexidade. Finalmente as implementações com reconhecimento de *hardware* que exploram os processamentos em GPU, principalmente em controles que necessitam manter suas funções em execução contínua por um longo período significativo e sem nenhuma interrupção, ou seja, alta disponibilidade (HARTMANN, 2020).

A motivação é usufruir melhor do paralelismo dos processos para ganho computacional, através da utilização de um número maior defluxos de processamentos independentes, onde o núcleo do sistema operacional invocado pode passar inúmeras execuções a serem processadas em GPU. Para tal uso a arquitetura de dispositivo de computação unificada (*Compute Unified Device Architecture,* CUDA) possibilita a execução no *host* da unidade central de processamento (*Central Processing Unit*, CPU) integrando com a GPU (SANDERS, 2010).

O paralelismo de processos pode ser visto em controles de operações de usinas hidrelétricas, envolvendo a representação para simulação de circuitos elétricos em software (GAMMA, 1995). A simulação de um circuito elétrico necessita da operação matricial com custo computacional tipicamente da ordem de . A medida que o número de nós no circuito aumenta, o impacto no desempenho do DT é considerável, sendo necessária a busca de alternativas para reduzir o tempo computacional para realizar as atualizações do circuito elétrico. O tempo computacional gasto deve ser o menor possível, para proporcionar fidelidade na simulação, nos limites do hardware disponível.

As pesquisas nessa linha demonstram necessidade de crescentes requisitos computacionais para a próxima geração de aplicativos, combinados com as melhorias no desempenho por meio das necessidades de impulsionar a tecnologia, resultaram em ambientes de computação heterogêneos e no aumento do uso de GPUs como aceleradores. Essa tendência chegou até as plataformas de computação em nuvem e até mesmo aos principais fornecedores nos dias atuais, como serviços web da amazon (*Amazon Web Services*, AWS) e serviços de nuvem Azure da Microsoft. A principal vantagem da aceleração baseada em nuvem é a capacidade de escalar sob demanda à medida que os requisitos de computação aumentam devido a conjuntos de dados maiores ou devido a restrições de latência. As GPUs de uso geral aproveitam o paralelismo no estilo instrução única e dados múltiplos (*Single Instruction, Multiple Data,*  SIMD) e *thread* simples de instrução única (*Single Instruction, Multiple Thread*, SIMT) e estão em uso há mais de uma década. Estas tecnologias se tornaram uma plataforma bastante utilizada atualmente, especialmente para a implementação de aplicativos de aprendizado de máquina. Em CPUs/GPUs, o subsistema de *cache* é fixo (tamanho) sendo gerenciado automaticamente pelo *hardware*. Portanto, aplicativos com baixa localidade de dados e pouca reutilização de dados ou pequenos conjuntos de trabalho exibem baixo desempenho e baixa escalabilidade (SHEPOVALOV, 2020).

Os estudos nesta área aprofundaram devido ao fato de que os sistemas de energia são aumentados continuamente, e o número de conversores eletrônicos de potência requerem simulações em tempo real, grandes sistemas com pequenos intervalos de tempo de simulação (menos de 1s), tornaram-se um grande desafio para as plataformas atuais. O método de MNA possibilita o paralelismo computacional entre a solução de modelos de rede com uma matriz de admissão invariável e de componentes em tempo real. Onde as simulações generalizam as aplicações incluindo vários conversores eletrônicos. O desempenho e precisão da plataforma proposta é comparada e verificada *versus* os estudos de casos de simulação *off-line*, como exemplo tem-se máquinas de indução sendo simuladas (MIRZAHOSSEINI, 2019).

Shiri (SHIRI, 2019) demonstrou a utilização do método de MNA na área da eletrônica, onde através de técnicas de modelagem para o conversor estático de frequência, utilizou a técnica de MNA e outras técnicas. Sendo que todas as demais caracterizavam também a função de transferência de entrada e saída. A MNA foi capaz de prever os diferentes picos de frequência, no entanto, estes picos tiveram a amplitude diminuída quando em baixa frequência (SHIRI, 2019).

A obtenção de equação de barramento elétrico industrial de equipamento, utilizado para a distribuição eficiente de energia elétrica em indústrias e possível pelo método de MNA. Suas aplicações ocorrem dentro dos quadros de distribuição de energia elétrica em subestações ou mesmo em aparelhos elétricos variados como máquinas industriais. A abordagem por este método de matriz pode reduzir o tempo necessário para derivar a equação da impedância (WANG, 2019).

O resultado obtido do cálculo pelo método da MNA demonstra ser mais preciso quando o processo utiliza a soma dupla tradicional com capacitores discretos, permitindo assim, criar um modelo matemático que possa suprimir ressonâncias no barramento de força. A execução ocorre com o uso de software de simulação de eletromagnética, com o intuito de verificar esses modelos, sendo utilizada a programação paralela para acelerar a velocidade de cálculo da matriz. Sendo assim, a matriz de impedância pode ser obtida através do método da MNA, através da combinação da soma dupla para calcular a impedância e dissociação dos capacitores para utilização simultânea (WANG, 2019).

A resolução da MNA é possível através de métodos diretos e iterativos. Como alternativa à técnica de decomposição inferior e superior (*Lower and* *Upper,* LU) usadas no circuito MNA podem ser usados outros métodos, como o método de decomposição superior e inferior (*Upper and Lower*, UaL). O método UaL tem demonstrado ser computacionalmente mais eficiente e menos propenso a erros de arredondamento quando comparados com a decomposição LU tradicional. O método UaL é desenvolvido para três casos de análise de circuitos. No primeiro para situações onde as fontes de excitação são todas as fontes atuais como tradicionalmente é o caso da representação do MNA. No segundo e no terceiro caso, são permitidas fontes de tensão de entrada no circuito, ou seja, fontes mistas. É demonstrado como uma combinação da representação de fontes pode generalizar todos os três casos em uma categoria. Em termos de precisão devido à capacidade do método de encontrar os fatores comuns e removê-los antes, todas as divisões com remanescentes são categoricamente eliminadas. E assim, removendo possíveis erros de arredondamento para os dispositivos (HASHEMIAN, 2019).

Outra abordagem para o uso do método MNA é na solução de rede de distribuição de eletricidade, principalmente quando abrange energia renovável. Um desafio significativo no caminho dessas redes de distribuição tem sido: planejamento, projeção e operação. O gerenciamento requer fluxo de energia rápido, análise, estimativa, suporte de potência reativa e outros. Sendo então proposto um método de análise de fluxo de potência que incorpore as características dos geradores distribuídos, demandando gerenciamento paralelo. A abordagem proposta reformulou a matriz para o método de MNA melhorando assim a solvabilidade e robustez de convergência, onde vários geradores distribuídos foram integrados e possibilitaram fornecimento de tensões para outros serviços auxiliares. Com o método proposto, o gargalo associado ao ajuste durante a atualização das potências reativas dos geradores distribuídos é completamente evitado (NDUKA, 2019).

Os dispositivos requerem seus equivalentes em circuitos quando possuam fontes de tensão. Essas equivalências são benéficas em termos de aplicabilidade na construção de projetos, sempre que precisem envolver uma gama maior de circuitos. Para este fim, faz-se necessário utilizar *softwares* especializados como o simulador de circuito eletrônico analógico (*Simulation Program with Integrated Circuit Emphasis,*SPICE). A partir desses circuitos equivalentes, pode-se elaborar ou simulações de sistemas de equações para circuitos analógicos baseados em vários métodos como a NA e MNA (ASANACHE, 2019).

A simulação permite levar em consideração cargas não lineares como os diodos, que são elementos básicos da eletrônica. Com base na teoria clássica das linhas de transmissão a rede é convertida em um circuito equivalente. O circuito é simulado usando um simulador de circuito escrito em MATLAB que é baseado na MNA. O modelo é validado em relação a uma solução existente no domínio da frequência para cargas lineares, os resultados são demonstrados em uma rede de três fios com cargas não lineares diferentes em paralelo. Assim, ocorre a integração com um simulador de circuito simples baseado em MNA no MATLAB, onde tem demonstrado ser mais simples calcular campos transitórios em diferentes locais espaciais (MAGDOWSKI, 2019).

Simulações hidrodinâmicas computacionais (*Computational Hydrodynamic Simulations,*CHS) constituem uma ferramenta de modelagem útil para aplicações de engenharia hidráulica. Dentro do contexto atual, é comum que simulações de CHS em larga escala sejam realizadas com técnicas de computação paralela de alto desempenho, para obter tempo de execução e simulação razoável para avaliação. A abordagem CHS adota esquemas numéricos apropriados para discretizar a massa e fluxos de energia com base nas equações de Navier-Stokes (NS). A cada interação enormes volumes de dados estão sendo compartilhados, o que destaca a importância de seus algoritmos de computação paralela inerentes para alcançar desempenho de aceleração desejável. Para acelerar simulações de CHS e simulações de fluxos a utilização de GPUs foram avaliadas (CHEW, 2020).

Outras áreas como na medicina, tem utilizado a MNA, para a simulação personalizada de doenças vasculares no cérebro, pode-se utilizar a MNA para modelagem agrupada de circuitos. Para a resolução de sistemas de equações discretas que resultem no acoplamento dos circuitos e das equações de campo eletromagnético. Em tal modelo as medidas hemodinâmicas foram calculadas usando uma versão adaptada da MNA (HO, 1975), onde simplificaram a solução das equações da matriz facilitando a implementação e resultando em tempo de execução mais rápido. A conservação do fluxo de massa pode ser aplicada usando equações para cada nó, e o conjunto de equações resultante pode ser resolvido para produzir as pressões arteriais e garantir o fornecimento constante de sangue para regiões específicas (FREY, 2020).

O fluxo sanguíneo foi modelado em formato de árvore em analogia aos circuitos elétricos em MNA. O sistema descrito é determinado por *n* equações, em que *n* representa o número de nós. O último nó é considerado o nó coletor, com valor de pressão 0. Portanto, o sistema é descrito pelas equações de fluxo de massa *n*-1. Essas equações de fluxo de massa podem ser escritas em termos de matriz, propiciando o desenvolvimento com implementação simplificada que reduz o custo computacional (FREY, 2020).

Para a paralelização automática baseada na solução de matrizes o resultado foi a formulação de equações de rede. Os métodos de NA, MNA e da simulação *off-line* são utilizados com o objetivo de obter paralelismo baseado em CPU, para computadores convencionais com vários núcleos usando soluções de matrizes esparsas. A MNA é usada para formular equações de rede e a solução da matriz esparsa selecionada é então paralelizada e adaptada para melhorar o desempenho, testando sua validade dinâmica e refatorando-a parcialmente. A refatoração é necessária quando é preciso lidar com redes de topologia variadas e modelos não lineares. A paralelização convencional, baseada em CPU do computador, é alcançada e não requer qualquer intervenção do usuário para determinadas topologias de rede arbitrárias. A abordagem apresentada é testada em tempo real em redes com modelos complexos, incluindo as não linearidades e conversores de dispositivos eletrônicos de potência (ABUSALAH, 2019). Ou seja, são esperados processamentos mais eficazes nas resoluções das equações de um circuito que possuam malhas e nós.

Assim o objetivo deste trabalho é propor alternativa à computação tradicional, empregando a MNA com 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 em desenvolvimento. Os modelos serão paralelizados e processados em placas gráficas para a redução temporal de simulação, onde o grande problema é o elevado custo computacional sendo processado sobre uma única unidade de processamento, resultando em um modelo menos eficiente. Para o cumprimento dos objetivos propostos, a metodologia utilizada visa explorar o uso de GPU para redução do tempo computacional envolvido na simulação do DT. Para esse propósito, emprega-se a arquitetura CUDA (MIVULE, 2008), que facilita o desenvolvimento de rotinas computacionais para processamento paralelo em GPUs utilizando linguagens de alto nível. Tal arquitetura tem encontrado diversas aplicações no âmbito da computação científica, podendo-se citar a solução de equações diferenciais estocásticas (JANUSZEWSKI, 2010), simulações de dinâmicas moleculares (LEE, 2008) e modelagem da poluição do ar (MOLNÁR, 2009) dentre outras. Como resultado, a expectativa e de que a taxa de crescimento do tempo computacional seja substancialmente menor na implementação usando CUDA quando comparada com uma implementação sequencial, utilizando a paralelização de algumas etapas envolvidas na simulação de um DT.

Para o experimento foi utilizada uma coleção de matrizes de operações da Usina Hidrelétrica de Jirau, referenciado como Gerador de Matrizes para *Modified Nodal Analysis*. É utilizada uma ferramenta *parser* (analisador) no processamento de linguagens, sendo responsável por verificar se a sequência de símbolos contida no código-fonte compõe um programa válido ou não. A análise de uma *netlist* do SPICE é o primeiro passo de todos os programas de simulação de circuitos. Esta parte normalmente é feita por técnicas de programação de baixo nível na linguagem C ou FORTRAN. O *parser* em C++ para um dialeto do SPICE modificado é capaz de reconhecer a sintática e realizar ações semânticas relevantes para as regras gramaticais com um baixo tempo de processamento permitindo assim separar as matrizes para o experimento de álgebra linear (SOUSA, 2019).

O presente trabalho tem sua estrutura construída em cinco capítulos. Além deste capítulo introdutório é apresentado uma visão geral sobre a trabalho proposto. No capítulo 2 apresenta-se a abordagem empregada na pesquisa, o objeto de estudo, a delimitação e o delineamento da pesquisa. No capítulo 3 são abordados os materiais e métodos relacionados com a arquitetura da GPU. No capítulo 4 demostramos o resultado do uso da programação paralela em CUDA para a multiplicação e inversão de matrizes. E finalmente, no capítulo 5 são apresentadas as considerações finais do trabalho analisando os resultados obtidos.

# Capítulo 2 – Sistemas de equações lineares

Este capítulo apresenta a fundamentação teórica utilizada no desenvolvimento deste trabalho. Os sistemas de equações lineares surgem em praticamente todas as áreas da matemática aplicada. O grande desenvolvimento dos meios de cálculo automático a que se tem assistido ultimamente tem possibilitado resolver sistemas com cada vez maior número de variáveis e de equações.

## 2.1 – Análise Nodal

A análise nodal (NA) 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 têm resistência zero e todos os pontos ligados a um nó tem a mesma tensão elétrica (ALEXANDER, 2017).

A NA 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 nós com um potencial fixo em relação ao nó de referência escolhido. Cada um dos nós têm uma equação associada para realizar a análise do circuito (ALBA-FLORES, 2013). 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 (SIMÕES, 2016).

Um Nó em um circuito, é o ponto de ligação de dois ou mais elementos, ou também, a união de dois ou mais condutores (SIMÕES, 2016). Em um Nó, é aplicado a lei de Kirchhoff, onde a somatória das correntes que entram, é igual a somatória das correntes que saem. As Leis de Kirchhoff descrevem o comportamento das tensões nas Malhas e das correntes nos Nós do Circuito. Sendo que uma Malha, é um caminho fechado de um Circuito ou qualquer caminho fechado de um condutor (O'MALLEY, 2011). Do conceito de Malha e de Nó, temos a Lei das Malhas de Kirchhoff (*Kirchhoff Voltage Law* - KVL) e a Lei dos Nós de Kirchhoff (*Kirchhoff Current Law*, KCL) (BOYLESTAD, 2015).

Nos métodos de análise de circuitos que envolvem as Leis de Kirchhoff são utilizados conceitos matemáticos, como a multiplicação e a inversão de matrizes. Essas operações matemáticas são necessárias para resolver os sistemas lineares. A resolução de sistema linear gera custo computacional elevado para resolver a inversão das matrizes. O resultado dessas equações soluciona o comportamento do Circuito a partir das variáveis: tensão, corrente e impedância elétrica (IRWIN, 2015).

2.1.1 – Análise Nodal Modificada

Uma análise nodal modificada (MNA) é um método que retém a simplicidade, onde uma das vantagens em relação a NA é remover algumas das suas limitações. Um esquema eficaz de inverter a matriz também é possível de ser obtido e assim através de valores numéricos é possível comparar o método MNA com o método NA. Lembrando que o conceito de matriz tem o formato *m* x *n* (leia-se: *m* por *n*, com *m* e *n* ∈ N\*), onde *m* é o número de linhas e *n* o número de colunas. Uma matriz é um agrupamento retangular de números e este agrupamento tem a denominação de entradas da matriz. O método MNA apresenta melhores resultados em termos de dimensão e entradas para matrizes de circuitos comparáveis (HO, 1975).

Para Ruehli (HO, 1975) a abordagem da NA clássica pode ser estendida de modo que as desvantagens são removidas e o seu uso se torna completamente geral. Uma desvantagem é o tempo de instalação gasto e as quantidades de armazenamento necessárias para a inversão da matriz, ponteiro de matriz e da geração de código. São consideravelmente reduzidos pela MNA em comparação com uma abordagem tradicional (HO, 1975).

**Tabela 1:** Comparação da abordagem com eliminação Gaussiana x métodos clássicos

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Caso | Método de Análise | Método de decomposição LU | | Comprimento do código (bytes) | | |
| 1 | Nodal  - OPTORD | Preenchido | Armazenado | T-Sol | X-Sol | B-Sol |
| 215 | 348 | 2200  1564 | 1868  1656 | 7864  6754 |
| 2 | Nodal  - OPTORD | 194 | 390 | 2812  1790 | 2388  1932 | 7668  6786 |
| 3 | S – V  - OPTORD | 472 | 426 | 4776  3968 | 2828  2630 | 9584  8288 |
| 4 | S – V  - OPTORD | 500 | 580 | 4044  3150 | 14492  14102 | 10620  9414 |
| 5 | NDP  - OPTORD | 191 | 344 | 2576  1652 | 2048  1686 | 7632  6554 |
| 6 | NDP  - OPTORD | 204 | 329 | 2820  1956 | 1572  1400 | 7792  6616 |

**Fonte:** Adaptado de Alexander (HACHTEL, 1971).

Legenda: T-solve, X-solve e B-solve correspondem a algoritmos.

O código SOLVE (HACHTEL, 1971) pode exigir muitos acessos aos núcleos do computador, sendo que a eliminação Gaussiana recursiva, método de matriz esparsa sugerido por Chang (HO, 1975) deve ser utilizado. Na Tabela 1 temos a comparação da abordagem entre a eliminação Gaussiana x Métodos Clássicos, através da utilização de pacotes de rotinas matriciais esparsas como OPTORD e 1-2-3 GNSO. Essas rotinas são responsáveis ​​pelo tipo de variabilidade dos elementos presentes na matriz, para a produção de código de programação para a solução da equação linear *Ax = b*. Iterações sequencias paralelas produzem as operações de soma ponderada e o erro de arredondamento incorrido na otimização é minimizado. Nas rotinas matriciais emprega-se a inversão direta da matriz através do método de fatoração *LU*. A saída do OPTORD é uma ordem de rotação, ou seja, a ordem na qual as linhas e colunas são eliminadas na inversão. A ordem selecionada fornece uma otimização entre o mínimo erro de arredondamento e total de operações na execução da iteração dupla. A 1-2-3 da GNSO produz o desenvolvimento baseado na criação das listas de instruções para a resolução no SOLVE em três partições: B-SOLVE, a ser executada no início de cada etapa de otimização, ou seja, antes de inserir o loop de tempo, T-SOLVE, a ser executado no loop de tempo (GEAR, 1968); e X-SOLVE, a ser executado no loop de Newton (GEAR, 1968).

Para a comparação utilizou-se de um projeto de rede de computadores, um programa para otimização automatizada da rede que incorpora totalmente os métodos de matriz esparsa, também foi utilizado para a comparação a abordagem esparta de Tableu (NDP) e S-V OPTORD para pequenos valores de intervalos de tempo.

Toda a eliminação de variáveis ​​é feita pela eliminação de rotina gaussiana. A matriz esparsa empacota os métodos utilizados pela OPTORD e da l-2-3 da GNSO. Nesse sentido, a abordagem de eliminação em quadros da matriz gaussiana tende a ter um efeito simplificador na análise de rede não apenas computacionalmente, mas até certo ponto teoricamente. Para fins computacionais oferecem uma melhor combinação de precisão e eficiência. Nas colunas que compõe o comprimento do código em bytes temos que o número superior representa a saída da corrente do compilador da 1-2-3 GNSO e o número inferior representa as estimativas baseadas em um compilador ideal. Os resultados da Tabela 1 demonstram os efeitos de diferentes estratégias para minimizar o comprimento do código do tipo x (X-SOLVE e B-SOLVE). Em todos os casos observados sem restrições o OPTORD obteve o melhor desempenho com a NA. Nesse caso como o método MNA gera uma matriz muito menor e com menos zeros, sendo muito mais rápida do que o método tradicional e do que a própria NA, temos que a velocidade de execução é diretamente proporcional às dimensões e não-zeros da matriz.

2.1.2 – Análise de malhas

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 (SALAM, 2018).

A análise de malhas consiste nas etapas descritas a seguir. Verificar a planaridade do circuito, caso o circuito não seja planar não é possível aplicar a análise de malhas. Escolher de modo arbitrário, o sentido das correntes da malha. O número de correntes arbitrárias deve ser igual ao número de ramos mais o número de nós do circuito (KANOUSSIS, 2017).

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 (VLACH, 2020).

## 2.2 – Algoritmos solucionadores de equações lineares

O problema é expresso matematicamente por variáveis, queremos resolver o sistema linear de *n* equações com *n* incógnitas , onde a existência da solução tem um determinante diferente de 0. Para isso temos algoritmos e regras definidas. Os métodos numéricos destinados a resolver sistemas lineares são divididos em dois grupos, os métodos diretos e os métodos iterativos.

Os métodos diretos produzem a solução exata de um sistema, a menos erros de arredondamento, após um número finito de operações aritméticas. Serão analisados os seguintes métodos diretos: Regra de Cramer, Eliminação Gaussiana, Decomposição LU, Decomposição QR e Decomposição de Cholesky.

Os métodos iterativos diferentemente dos métodos diretos, não encontram a solução diretamente, e sim gerando uma sequência de vetores , através de uma fórmula recursiva, dada uma aproximação inicial que pode ou não convergir para a solução, caso exista. Quando convergente, a sequência tenderá à solução quando *k → ∞*. Como é uma fórmula recursiva o termo seguinte depende do termo anterior, sendo necessário determinar o vetor que iniciará esta sequência. Serão analisados os seguintes métodos indiretos: Gauss Jacobi, Gauss Seidel e Valores e vetores próprios para matrizes simétricas.

2.2.1 – Regra de Cramer

A regra de Cramer é um [teorema](https://pt.wikipedia.org/wiki/Teorema) em [álgebra linear](https://pt.wikipedia.org/wiki/%C3%81lgebra_linear), que dá a solução de um [sistema de equações lineares](https://pt.wikipedia.org/wiki/Sistema_de_equa%C3%A7%C3%B5es_lineares) em termos de [determinantes](https://pt.wikipedia.org/wiki/Determinante) (KOSINSKI, 2001). Recebe este nome em homenagem a [Gabriel Cramer](https://pt.wikipedia.org/wiki/Gabriel_Cramer) (1704 - 1752). O processo para a regra de Cramer é verificar se o [sistema linear](https://pt.wikipedia.org/wiki/Sistema_de_equa%C3%A7%C3%B5es_lineares) é determinado, indeterminado ou impossível. Isto é um procedimento necessário, pois o *software* somente processará a resolução do sistema linear caso ele seja possível e determinado. A Regra de Cramer nos dá uma fórmula para resolver sistemas lineares do tipo:

(1)

Em termos de determinantes. A regra nos fornece instruções precisas para o cálculo das incógnitas em função dos coeficientes e dos termos . Observe que o sistema acima consiste de *n* equações lineares com ... . Assim, o número de equações é igual ao número de incógnitas. A fórmula de Cramer só se aplica a sistemas lineares dessa espécie. Em termos de matrizes, a Equação (1) pode ser escrito como:

(2)

Onde:

(3)

2.2.2 – Eliminação Gaussiana

Este procedimento de álgebra linear primeiro realiza uma eliminação progressiva. A eliminação gaussiana reduz um determinado sistema a um sistema triangular. Segundo, realiza uma eliminação regressiva para resolver o sistema linear (LYCHE, 2020).

Primeiro passo: eliminar nas linhas: 2 até *n* por combinação linear de linhas. Segundo passo: eliminar nas linhas: 3 até *n* por combinação linear de linhas. Iterativamente após n-1 passos. Obtendo assim o algoritmo de eliminação gaussiano pela eliminação progressiva e forma triangular.

(4)

Por eliminação regressiva, significa que nós começamos por eliminar nas linhas. Iniciando de 1 até n-1 por combinação linear de linhas. Pela iteração deste processo a matriz só tem números na diagonal principal e assim obtemos a solução x. No Apêndice A.1 consta demonstração da solução através de exemplo prático.

2.2.3 – Decomposição LU

A decomposição LU é um procedimento para simplificar uma matriz. Decomposição *LU* é frequentemente usado na solução de problemas lineares em computadores. Uma forma de fatoração de uma [matriz](https://pt.wikipedia.org/wiki/Matriz_(matem%C3%A1tica)) não singular como o produto de uma [matriz triangular](https://pt.wikipedia.org/wiki/Matriz_triangular) inferior (*lower*) e uma matriz triangular superior (*upper*). Às vezes se deve pré-multiplicar a matriz a ser decomposta por uma matriz de permutação. Esta decomposição se usa em [análise numérica](https://pt.wikipedia.org/wiki/An%C3%A1lise_num%C3%A9rica) para resolver sistemas de equações (mais eficientemente) ou encontrar as matrizes inversas (RAHMANI, 2020).

Para uma solução geral do sistema linear e decomposição LU de uma matriz geral *m x n*, utiliza-se a matriz *A* como pivotação parcial com trocas de linhas:

(5)

Onde *P* é uma matriz de permutação, *L* é a triangular inferior com unidade diagonal e *U* é a diagonal superior. As informações nas linhas intercambiadas ficam contidas em uma variável, determinda como pivo. Através da fatoração obtida é possível substituir o problema e resolver o sistema linear geral, resolvendo dois sistemas triangulares com matrizes *L* e *U*, respectivamente.

(6)

*L* = *U* = (7)

Dado o sistema da Equação (6) resolvemos a Equação (6a) para encontrar o vetor *y*, depois a Equação (6b) para encontrar o vetor *x*. A resolução é facilitada pela forma triangular das matrizes.

Antes de iniciar a Equação deve ser encontrada a solução *L* e *U* da Equação (7), que normalmente são resolvidos pela eliminação Gaussiana. No Apêndice A.2 consta demonstração da solução através de exemplo prático.

2.2.4 – Decomposição QR

Para a decomposição QR e verificação da ortogonalidade, o método de Gram-Schmidt permite fatorar uma matriz *A* cujas colunas são linearmente independentes em um produto (NAEINI, 2020).

(8)

Onde na Equação (8) *Q* é uma matriz cujas colunas formam um conjunto ortonormal e *R* é uma matriz triangular superior. Tal decomposição é muito importante do ponto de vista computacional, pois favorece os processos paralelizados, ao usarmos a decomposição *QR* como uma alternativa para resolução de sistemas lineares por mínimos quadrados. Também é a base de um eﬁciente método numérico para encontrar autovalores de uma matriz quadrada. A decomposição *QR* está para o processo de Gram-Schmidt assim como a decomposição LU está para o método de eliminação gaussiana.

A base ortonormal gerada pelo processo de Gram-Schmidt será sempre invertível se sua inversa for dada pela sua transposta, ou seja *−1 =T*. Assim temos que se, basta multiplicar ambos os lados desta equação por*T* pela esquerda para obtermos:

*TTT* (9)

No Apêndice A.3 consta demonstração da solução através de exemplo prático.

2.2.5 – Decomposição de Cholesky

A decomposição Cholesky é um método de álgebra linear bem conhecido para a decomposição matricial. Descoberto por Andrew Louis Cholesky que afirma que qualquer matriz que seja simétrica e de definição positiva, pode ser decomposta em uma matriz triangular inferior. Onde dada uma matriz positiva simétrica *A* o objetivo é construir uma matriz triangular inferior *L* que tem a seguinte propriedade: o produto de *L* e a sua transposição é igual a *A* (LEW-YEE, 2020).

(10)

Onde através da decomposição Cholesky (Equação 10) obtém-se a fatoração para decompor a matriz *A* em para a resolução de sistema linear:

(11)

Para a resolução do sistema linear (Equação 11) na forma:

(12)

Resolve-se a Equação (a) para encontrar o vetor , depois a Equação (b) para encontrar o vetor . A resolução é facilitada pela forma triangular das matrizes. No Apêndice A.4 consta demonstração da solução através de exemplo prático.

2.2.6 – Gauss Jacobi

O método iterativo Gauss Jacobi obtém uma aproximação para a solução de um sistema linear. Geralmente em um método iterativo, inicia-se com uma aproximação para a solução e este tende a melhorar, conforme ocorre a aproximação através de sucessivas iterações (GASSIAT, 2020).

Considere um sistema linear , em que é uma matriz não-singular com *aii* ≠ 0, ∀*i* = 1, . . . , *n*. Podemos escrever o sistema da seguinte forma:

(13)

Dada uma aproximação inicial (0) para a solução do sistema , o método de Jacobi define a sequência de vetores {(k)}k ≥ 0 através da relação de recorrência (Equação 13).

Observa-se que, a cada linha, é necessária a divisão pelo termo da diagonal principal correspondente a respectiva linha. Desta forma, é necessário que nenhum termo da diagonal principal seja nulo. Sendo possível ocorrer inúmeras trocas entre linhas ou colunas, para adequar a matriz. No Apêndice A.5 consta demonstração da solução através de exemplo prático.

2.2.7 – Gauss Seidel

A estrutura do método de Gauss Seidel é análoga à de Gauss Jacobi. No entanto, quando ocorre o cálculo do termo *j* (*k* + 1), o processo utiliza os termos *i* (*k* + 1) já calculados previamente, para os valores de *i* tais que *i* ∈{1, . . . , *j* − 1*}*, e *i* (*k*), para os valores de *i* tais que *i* ∈ {*j* + 1, . . . , *n*} (DING, 2020).

Assim, a equação recursiva é dada por:

14)

A estrutura da equação recursiva de iteração é similar à de Gauss Jacobi, o número de operações em uma iteração também é o mesmo conforme Equação (14). Assim, temos um total de 2*n*2 *– n* operações em uma iteração. Para o formato matricial deve-se dividir a matriz *A* do sistema em 3 partes:

-A matriz *L*, triangular inferior, com diagonal nula, que corresponde aos termos que serão aplicados ao vetor *x*(*k*+1);

- A matriz *D*, matriz diagonal, com os termos da diagonal principal diferentes de 0, e os demais termos iguais a 0;

-A matriz *R*, triangular superior, com diagonal nula, que corresponde aos termos que serão aplicados ao vetor *x*(*k*).

(15)

A representação matricial pelo método de Gauss Seidel para uma matriz inicial *A,* conforme a Equação (15). Para a garantia de convergência do método de Gauss Seidel temos que verificar alguns critérios:

- O critério de Sassenfeld for satisfeito, ou seja,

seja *max*1≤i≤n *βi* < 1.

em que *βi =*

- O critério das linhas for satisfeito, ou seja,

seja *max*1≤i≤n

- Se a matriz dos coeficientes for estritamente diagonal dominante.

No Apêndice A.6 consta demonstração da solução através de exemplo prático.

2.2.8 – Valores e vetores próprios para matrizes simétricas (SVD)

Seja *A* uma matriz quadrada de ordem *(n* x *n).* Um autovalor de *A* um número real r, tal que, subtraído de todos os elementos da sua diagonal, converte *A* numa matriz singular (PENNEC, 2020).

(16)

*A' = A* – r\*I com = 0. (17)

**Teorema 1**: Os elementos da diagonal de qualquer matriz diagonal são seus autovalores (Equação 16).

**Teorema 2**: Uma matriz quadrada é singular se, e somente se o valor 0 é um dos seus autovalores (Equação 17).

Em geral, matrizes quadradas de ordem *(n* x *n)* tem no máximo *n* autovalores reais. Contando as raízes repetidas e complexas do polinômio característico, tem-se que matrizes de ordem *(n* x *n)* possuem exatamente *n* autovalores.

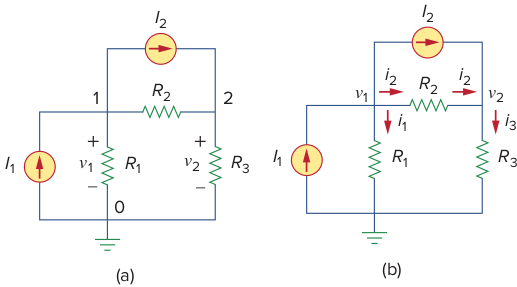
Seja **r** um autovalor qualquer de uma matriz quadrada *A*. Um vetor v ≠ 0, tal que (A-r\*I) v = 0, denomina-se o Autovetor correspondente ao Autovalor **r**. Sendo *B* uma matriz quadrada singular, então *Bx = 0* possui infinitas soluções para o vetor x. Portanto, para qualquer matriz quadrada *A*, se **r** é um dos seus Autovalores, sempre existe um Autovetor correspondente a esse Autovalor. No Apêndice A.7 consta demonstração da solução através de exemplo prático.

## 2.3 – Circuito para análise nodal

O principal problema a ser superado no contexto deste trabalho é a solução dos sistemas de equações lineares para obter as tensões dos nós dos circuitos elétricos que são processados pelo módulo do fluxo de potência do DT em desenvolvimento 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.

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 modificada porque a análise de malha só é utilizada para circuitos planos. O ambiente DT requer a análise de circuitos não planos.

Figura - Circuito para análise nodal

 Fonte: Adaptado de Alexander (ALEXANDER, 2017)

O objetivo da NA é encontrar as tensões dos nós. Assim, dado um circuito com 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 tensõespara os restantes (SMITH, 2019). As tensões são referenciadas em relação ao nó de referência; 2) Aplique a KCL a cada um dos 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 as tensões conhecidas dos nós (ROBBINS, 2012).

## 2.4 – Resolução de circuitos pela análise nodal modificada

O primeiro passo da MNA 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 aos nós não referência.

No circuito da Figura 1(a), o nó 0 é o nó de referência (). Os nós 1 e 2 recebem as tensões e respectivamente. As tensões do nó são definidas em relação ao nó de referência (THOMAS, 2011). 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 MNA é feita aplicação da KCL a cada nó não referência do circuito. A Figura 1(b) mostra que e não são somados como as correntes através dos resistores e respectivamente. A aplicação da KCL para o nó 1 obtém-se

(18)

Até o nó 2, tem-se

(19)

Nessa etapa é aplicada a lei de Ohm para expressar as correntes desconhecidas e em termos das tensões nos nós (HAYT, 2018). 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:

. (20)

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

ou

ou (21)

ou

Substituindo a Equação (21) na Equação (18) e Equação (19), tem-se, respectivamente:

(22)

(23)

Em relação às condutâncias, Equação (22) e (23), tem-se:

(24)

(25)

A terceira etapa da MNA é a resolução das tensões dos nós. Aplica-se a KCL aos nós não referência. Obtém-se equações simultâneas, como Equações (22) e (23) ou (24) e (25) (NAHVI, 2017). Para o circuito da Figura 1, a solução das Equações (22) e (23) ou (24) e (25) produz as tensões dos nós e usando qualquer método padrão, como o método de substituição, o método de eliminação, a regra de Cramer ou a inversão da matriz (ZILL, 2016). Para usar um dos dois últimos métodos, é necessário converter as equações simultâneas na forma de matricial. As Equações (24) e (25) podem ser escritas na forma matricial:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |  |  | = |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | (26) |

Considerando o circuito da Figura 1(a) para o uso da MNA e ganho computacional, tem se a resolução do sistema linear 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 *Application programming interface* (API) CUDA. A definição por usar núcleos da API CUDA é feita pela declaração dos recursos invocados explicitamente*.* 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.

2.4.1 – Custo computacional dos algoritmos

Em vista da existência de características diferentes dos métodos diretos e iterativos, foram analisados os mais eficientes e sua complexidade para resolver sistemas lineares. Analisado as principais características teóricas pertinentes de tais métodos.

**Tabela 2:** Comparação dos métodos diretos e iterativos em relação a vários aspectos

|  |  |  |
| --- | --- | --- |
| Aspectos | Métodos diretos | Métodos iterativos |
| Convergência | A solução é sempre obtida. | Há garantia de obter solução somente sob certas condições. |
| Número de operações | É possível determinar a priori o número de operações necessárias. | Não é possível determinar a princípio a complexidade. |
| Esparsidade | Destrói a esparsidade da matriz durante a fase de eliminação. | Preserva a esparsidade da matriz. |
| Erros de arrendondamento | Os erros aparecem durante a aplicação das transformações elementares sobre as equações do sistema. Esse erro se propaga porque as transformações de cada fase são realizadas a partir de equações transformadas da fase anterior, as quais estão sujeitas a erros. Essa propagação de erros pode ser minimizada usando-se técnicas de pivoteamento. | Apenas a solução corrente é sujeita a erro. Não há propagação dos erros porque são usados os coeficientes originais da matriz *A* e vetor *b* no cálculo das componentes da solução. |

**Fonte:** Fonte: Elaborada pelos autores (2020)

A simulação de tais modelos gera um custo computacional elevado, o que justifica a investigação de implementações em plataformas de processamento paralelo. Neste projeto, será explorado o uso de unidade de processamento gráfico (*Graphics Processor Unit*, GPU) empregando a arquitetura de dispositivo de computação unificada (*Compute Unified Device Architecture*, CUDA), desenvolvida pela NVIDIA. Sendo utilizado através da computação paralela e uso de extensão para a linguagem de programação C++, e o apoio de unidade de processamento gráfico (GPU) (CUDA, 2020).

## 2.5 – Considerações do Capítulo

Neste capítulo foi comentando sobre a fundamentação teórica da análise nodal modificada e consequente dos sistemas de equações lineares. Comentado sobre os métodos diretos e iterativos. Iniciando pela Regra de Cramer, Eliminação Gaussiana, Decomposição LU, Decomposição QR, Decomposição de Cholesky, Gauss Jacobi, Gauss Seidel e SVD. Explicado sobre a análise de circuito implementada para a tecnologia GPU CUDA. Resumo do custo computacional dos algoritmos pelos métodos diretos e iterativos.

# Capítulo 3 – Programação Paralela

Este capítulo apresenta prospectos sobre computação paralela. Explica o que é o CUDA e o que é mais importante nas suas características.

## 3.1 – Capacidade de processamento

*FLOPS* (flops, FLOP/s ou flop/s) é um acrônimo na computação que significa operações de ponto flutuante por segundo (*Floating-point Operations Per Second)*. É um valor utilizado para indicar a taxa de velocidade dos microprocessadores (em computadores). Expressa a performance do microprocessador. Os dispositivos de computação têm enorme capacidade de processamento, convém utilizar unidades maiores que *FLOPS*, seus múltiplos. Os múltiplos mais utilizados são: megaflop/s (Mflop/s), gigaflop/s (Gflop/s), teraflop/s (Tflop/s), petaflop/s (Pflop/s) e exaflop/s (Eflop/s).

As representações de valores são expressas para facilitar a demonstração numérica do desempenho de um sistema com supercomputadores, é necessário escolher uma métrica de performance que possa representá-lo sem introduzir distorções, sem influenciar os resultados e sem favorecer um determinado escalonador do computador (FEITELSON, 2011).

**Tabela 3:** Valores para demonstrar a quantificação do desempenho computacional

|  |  |
| --- | --- |
| **Desempenho Computacional** | |
| Ordem de grandeza | Quantidade (flop/s) |
| megaflop/s | 106 |
| gigaflop/s | 109 |
| teraflop/s | 1012 |
| petaflop/s | 1015 |
| exaflop/s | 1018 |
| zettaflop/s | 1021 |
| yottaflop/s | 1024 |

**Fonte:** (GWENNAP, 2011)

Na Tabela 3 é demonstrado as representações de valores de cada um dos múltiplos dos *FLOPS* e os seus respectivos valores.

3.1.2 – Lei de Amdahl

A Lei de Amdahl (AMDAHL, 1967) é um dos métodos que tentam explicar o ganho obtido em uma aplicação ao adicionar mais processadores. O norueguês Gene Amdahl iníciou suas pesquisas seguindo a ideia de que um trecho do programa é puramente sequencial. Sua apresentação inicial ocorreu na Conferência Conjunta de Informática unificada (*Conference Proceedings*, AFIPS) no ano de 1967 (GUSTAFSON, 1988).

A noção de aceleração estabelecida pela lei de Amdahl, foi particularmente focada no processamento paralelo, objetivado na arquitetura de computadores e no aumento de velocidade. O aumento de velocidade pode ser usado de forma mais geral para mostrar o efeito no desempenho após qualquer aprimoramento de recursos (AMDAHL, 1967).

Amdahl definiu que o valor do *Speedup* está limitado pela parte do programa inerentemente sequencial. Onde temos o tempo para executar o programa em um único processador (*Ts*), não paralelizável. E uma parte susceptível de ser paralelizável, o tempo para executar o programa em vários processadores paralelos (*Tp*), que pode ser executado com *Speedup* *P* num computador de *P* processadores (AMDAHL, 1967).

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Figura 2 - *SpeedUp* | | | | |  |
|  | *Speedup* | *=* | *Ts + Tp* |  |
|  | *Ts +* |  |

Fonte: Amdahl (AMDAHL, 1952)

Na Figura 2 o *Speedup* está definido por um processador paralelo com *P* processadores que explora totalmente a parte paralela do programa.

Figura 3 - *Speedup* proporção paralelizável

|  |  |  |  |
| --- | --- | --- | --- |
| *p* | *=* | *Tp* |  |
| *Ts + Tp* |  |

Fonte: (AMDAHL, 1952)

Na Figura 3 é definida a proporção do código paralelizável. A proporção *p* de código sequencial por *s =* 1 *– p*, então a precisão demostrada na Figura 2 e equivalente:

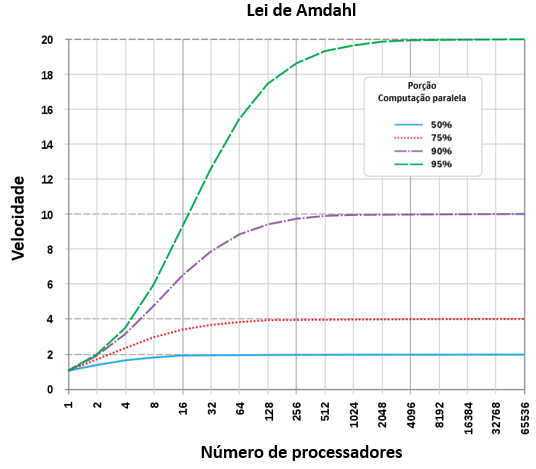
|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Figura 4 - *Speedup* calculado | | | | |  |
|  | *Speedup* | *=* | *1* |  |
|  | *s +* |  |

Fonte: Amdahl (AMDAHL, 1952)

Na Figura 4 é demonstrado o *Speedup* em equivalência. Como exemplo ao aferir um tempo de execução no programa sequencial de *Tp* = 90*s* e 93*s* de tempo total, obtém se *p* = 90/93, com um resultado igual 0.9677, ou seja 96.8% do código paralelizável, enquanto 3.2% e inerentemente sequencial. As operações sequenciais poderão ser de entrada de dados, saída de dados e inicialização de dados. Utilizando a equação da Figura 4 quando *P* → ∞ temos o *Speedup* = 1/s. Para o exemplo acima o *Speedup* = 1/0.032 = 31.25. Concluindo que não haverá qualquer vantagem em usar mais do que 31 processadores.

A Lei de Amdahl demonstra os problemas enfrentados pela indústria no desenvolvimento de máquinas *multicore* com um número cada vez maior de processadores: o software que roda nessas máquinas, deve ser adaptado para um ambiente de execução altamente paralelo, para explorar o poder do processamento paralelo (PATTERSON, 2016).

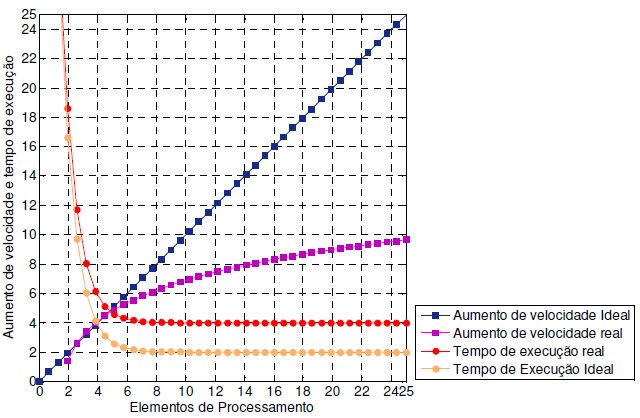
Figura - Argumento de Amdahl



Fonte: Amdahl (AMDAHL, 1952)

Na Figura 5 é demonstrado o gráfico para a justificativa da argumentação de *speedup* ligado ao aumento de processadores segundo Amdahl. A Lei de Amdahl, também conhecida como argumento de Amdahl é usada para encontrar a máxima melhora esperada para um sistema, em geral quando apenas uma única parte do mesmo é melhorada. Isso é frequentemente usado em computação paralela para prever o máximo *speedup* teórico usando múltiplos processadores (AMDAHL, 1952).

Figura - Tempo de execução e *speedup*



Fonte: Adaptado de Amdahl (AMDAHL, 1952)

Na Figura 6 é demonstrado o gráfico do tempo de execução em relação ao *speedup* de programa paralelizável. Onde a linha azul demonstra um cenário ideal, a linha lilás indica o aumento real, a linha amarela indica o tempo de execução no caso ideal e a linha vermelha o tempo real. A Lei de Gustafson dá uma avaliação mais realista do desempenho paralelo (GUSTAFSON, 1988).

3.1.3 – Lei de Gustafson

A análise de Amdahl parte do princípio de que a fração serial (não-paralelizável) do algoritmo é independente do número de processadores. Para Gustafson do ponto de vista prático tal consideração não seria razoável em alguns casos, pois as tarefas aumentam com o aumento da capacidade computacional e a fração de código serial, porém em sua grande maioria, não existe crescimento de mesma proporção. Isto levaria muitos algoritmos paralelos a terem sua fração serial reduzida com o aumento do número de processadores. A partir da observação de resultados práticos, Gustafson sugere uma métrica alternativa a Lei de Amdahl, onde um só processador é no pior dos casos poucas vezes mais lento que a execução com alguns processadores, não tendo assim grandes vantagens aparentes (GUSTAFSON, 1988).

3.1.4 – Speed-up

Ganho de velocidade (*speed-up*) consiste na comparação entre o tempo de execução do programa em um único processador e o tempo de execução utilizando vários processadores. O *speedup* de um programa utilizando a estrutura de múltiplos processadores em computação paralela tem limitações impostas pelo tempo dispensado para a execução de uma fração sequencial de um programa (AMDAHL, 1952).

Como exemplo, se o programa gasta o equivalente a 40 horas de alocação de um único núcleo de processamento, e a parte específica de uma rotina do programa gasta duas horas para executar não podendo ser paralelizada, enquanto as 38 horas restantes (que é equivalente a 95%) do tempo de processamento de execução pode ser paralelizado, independente de quantos processadores são dedicados a execução paralela deste programa, o tempo de execução mínima não pode ser menor que a seu maior processamento. Sendo assim, temos que o aumento de velocidade é limitado geralmente ao maior tempo de processo paralelizado (AMDAHL, 1967).

Para um programa que gaste 10% de seu tempo em um componente seqüencial, a aceleração máxima possível através da paralelização é 10. Devido a Lei de Amdahl, é necessário melhorar a desempenho tanto do seqüencial quanto do paralelo para execução de componentes (ANNAVARAM, 2005).

## 3.2 – Multiplicação de matrizes

O processo aplicado na multiplicação de matrizes consiste da multiplicação da matriz *Am*×*n* pela multiplicação da matriz *Bn*×*w*. O resultado dessa operação consiste na geração de uma nova matriz *Cm*×*w*, sendo assim temos *A* × *B* = *C*. Na Figura 7 é demonstrado o objetivo a ser alcançado pela multiplicação de matrizes, que é a obtenção destes elementos em uma nova matriz denominada de *C*. Onde cada linha de *A* é multiplicada elemento a elemento por uma coluna de *B*. A consequência a ser concluída é o valor da multiplicação por meio de somatório. É uma premissa que a multiplicação seja executada com a mesma quantidade de colunas da matriz *A* igual a mesma quantidade de linhas da matriz *B*.

Figura - Demonstração de multiplicação de matrizes



Fonte: Adaptado de Pacheco (PACHECO, 2019)

Como os elementos de *C* são computados de maneira independente, temos que eles podem ser paralelizados na multiplicação de matrizes. Sendo desta forma elegíveis como bons para a aplicação da computação paralela.

## 3.3 – Abordagem de programação paralela

As principais dificuldades consistem em desenvolver, gerenciar e manter o sistema em virtude da alta complexidade encontrada neste tipo de cenário, controlar o acesso concorrente a dados e a recursos compartilhados, evitar que falhas de máquinas ou da rede comprometam o funcionamento do sistema, garantir a segurança do sistema e o sigilo dos dados trocados entre máquinas e lidar com a heterogeneidade do ambiente.

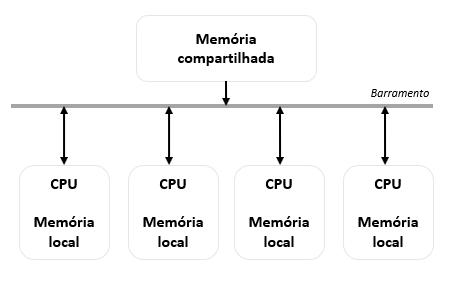
Algumas das vantagens embutidas com a programação paralela consistem em sistemas paralelos fortemente acoplados, compartilhando hardware ou se comunicando através de um barramento de alta velocidade, previsibilidade em virtude do comportamento de sistemas paralelos ser bastante previsível em comparação com outros tipos de programação e o controle de todos os recursos computacionais.

Existem estudos que desenvolveram trabalhos para avaliação e medição do desempenho das principais APIs de programação paralela, processando a execução desses algoritmos em CPUs e GPUs. Para que tal avaliação de desempenho ou ganho computacional ocorra inicialmente é necessário a instalação em computador das APIs CUDA (GPU), OpenCL (GPU), OpenACC (GPU) e OpenMP (CPU) (CORRÊA, 2017). Os *speedups* desses modelos de trabalhos mostraram que a GPU possibilita desempenho mais elevado em comparação a CPU na maioria dos algoritmos estudados. Tal fato e creditado pela GPU possuir mais núcleos em relação a CPU, mesmo tendo frequência dos núcleos inferiores (CORRÊA, 2017). Os resultados demostram que os processos paralelos têm maior ganho computacional conforme a complexidade dos algoritmos ou mesmo do processo computacional.

Em geral a paralelização direta das aplicações normalmente satura a largura de banda de memória (DRAM), resultando em um ganho de velocidade pequeno. O grande trunfo e contornar as limitações da largura de banda de memória, o que envolve fazer uma das muitas transformações para utilizar as memórias especializadas no chip GPU, para reduzir e otimizar os acessos a DRAM. Se a aplicação inclui o chamado paralelismo de dados, geralmente atinge-se um ganho de velocidade de 10x com apenas algumas horas de trabalho.

Um Sistema operacional multitarefa permite simular o paralelismo em um único processador, alternando a execução de processos, um processador com núcleo múltiplo permite paralelismo real entre processos, executando múltiplas instruções por ciclo, uma placa mãe multiprocessador permite que cada processador execute um processo. Cluster é uma solução de baixo custo para processamento de alto desempenho, uma computação distribuída é possível em redes, como numa *Intranet* e na *Internet* (CUDA, 2020).

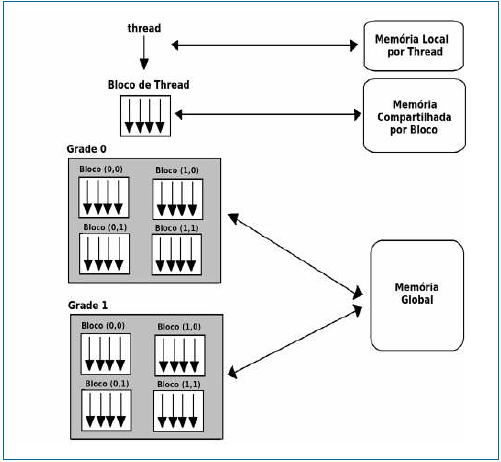
Figura - Sistema paralelo

Fonte: Adaptado de Correa (CORRÊA, 2017)

A Figura 8 demonstra o cenário de programa que possui um conjunto de instruções em uma linguagem de alto nível ou de máquina, um processo que resulta da execução do programa e *threads* que são processos menores que compartilham o mesmo "espaço de endereçamento“ (memória). Cada um dos multiprocessadores tem várias ALUs que em qualquer ciclo de *clock* executam as mesmas instruções, tendo assim uma quantidade enorme de capacidade de processamento.

Em síntese os principais pontos das memórias de uma GPU são compostos pela memória global, memória compartilhada e memória local. A memória global é a principal memória da GPU, podendo ser acessadas por *threads/cores,* porém possuem alta latência e baixo *throughput.* A memória compartilhada é a memória dedicada de cada SM (*Stream Multiprocessors*), as quais possuem baixa latência. Apenas *threads* de mesmo bloco podem ter acesso a ela. Já no caso da memória local, ela recebe este nome pois é a memória específica de uma *thread* (CHAPMAN, 2008).

Figura - Memória compartilhada



Fonte: Adaptado de Chapman (CHAPMAN, 2008)

Na Figura 9 o acesso as memórias pelas *threads* são descritas. Onde a CPU tem apenas o acesso a memória global. Assim a distribuição de *threads* e a hierarquia de memória também são demonstradas.

O paralelismo pode ser obtido de outras maneiras, das quais podemos citar a utilização dos núcleos de CPU, utilizando o multi-processamento aberto (*Open Multi-Processing*, OpenMP), que são interfaces que possibilitam o uso de programação paralela com memória compartilhada, usadas na arquitetura composta por múltiplos processadores (OPENMP, 2017).

3.3.1 – OpenMP

O OpenMP é composto por uma especificação que fornece modelo de programação paralela com compartilhamento de memória. Esta API possui um conjunto de diretivas as quais são integradas as linguagens C/C++ e Fortran, usufruindo do conceito de *threads*, contudo dispensa que o programador tenha que trabalhar diretamente ou exclusivamente com elas. Tal conjunto de diretivas sempre que acionadas e adequadamente configuradas criam blocos para serem paralelizados e distribuem o processamento entre os núcleos disponíveis ou acessíveis (OPENMP, 2017).

O programador não necessita ter grande preocupação em criar *threads* ou dividir as tarefas manualmente no seu código fonte. O OpenMP assume tal função e encarrega do processamento em alto nível (SENA, 2008).

O modelo de programação do OpenMP é conhecido por *fork-join*, onde um processo tem seu início como um único *thread* que executa sozinho todas as instruções até encontrar uma região que seja paralisável, sendo que esta é identificada por uma diretiva OpenMP. Ao determinar esta região é acionada um grupo de *threads* que são alocadas e juntas executam o código paralelizado. Ao encerrar esta execução do paralelismo os *threads* são sincronizados e deste ponto em diante, somente uma *thread* é que segue com a execução do código sequencial (MATLOFF, 2016).

A realização da paralelização por meio da CPU utiliza OpenMP. Essa API dá suporte às linguagens C/C++ e Fortran, onde é possível incluir ao código original a diretiva em trechos de códigos que podem ser paralelizados. A API dispara *threads* utilizando os núcleos da CPU, através de uma diretiva para paralelizar um loop na linguagem C++:

|  |  |  |
| --- | --- | --- |
|  | #pragma omp parallel for default(none) shared(n, x, y) private(i)  for (i=0; i<n; i++)  x[i] += y[i];  } |  |
|  |  |  |

A declaração *pragma omp* corresponde a funcionalidade básica de comunicar informação ao compilador de C/C++ de modo a este gerar código otimizado para o ambiente de execução do OpenMP.

A diretiva *parallel* é o construtor fundamental do OpenMP. *default(none)* define que o tipo de todas as variáveis envolvidas na região paralela deve ser declarado explicitamente (sobrepõe-se à definição de que, por omissão, as variáveis são consideradas partilhadas).

A cláusula *shared(list)* define sobre as variáveis definidas em *list* são partilhadas por todos os threads ficando à responsabilidade do programador garantir o seu correto manuseamento. Por omissão, as variáveis para as quais não é definido qualquer tipo são consideradas variáveis partilhadas.

A cláusula private(list) define que as variáveis definidas em *list* são duplicadas em cada *thread* e o seu acesso passa a ser local (privado) em cada thread. O valor inicial das variáveis privadas é indefinido (não é iniciado) e o valor final das variáveis originais (depois da região paralela) também é indefinido.

3.3.2 – OpenACC

O desenvolvimento do OpenACC tem sua origem da iniciativa e esforço de um grupo de empresas, onde podemos incluir e destacar principalmente a NVIDIA, Portland Group Inc, CAPS Enterprise e CRAY. O OpenACC possui módulos e especificações para executar os *softwares* desenvolvidos em C, C++ e Fortran a partir de uma arquitetura que utilize a CPU para que haja uma compatibilidade com um dispositivo acelerador (WIENKE, 2012).

Em virtude de ser composto por um conjunto de diretivas o OpenACC verifica a estrutura dos *softwares*, os quais compartilham informações, como o *host* (CPU) e o dispositivo acelerador. Desta etapa em diante é necessário um mapeamento otimizado sendo gerada execução por *cores* paralelos (OPENACC, 2015). Em regra, o objetivo principal da API é a aceleração, porém o OpenACC fornece também uma forma de migrar os aplicativos com algumas alterações na forma como o código sequencial do algoritmo esteja escrito (WIENKE, 2012).

3.3.3 – OpenCL

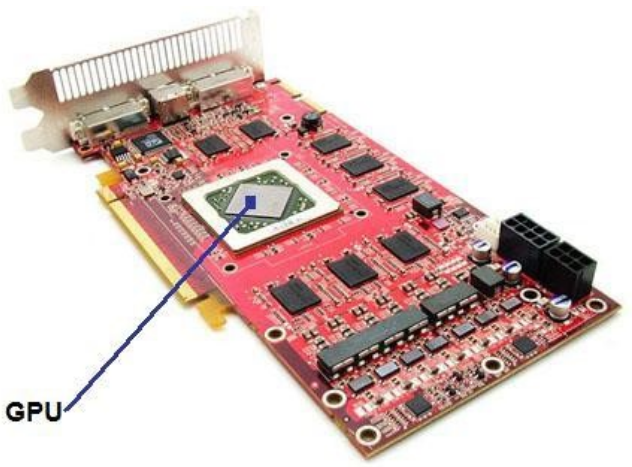
Como uma outra alternativa à plataforma OpenCL possibilita a codificação tanto para GPUs NVIDIA quanto para ATI/AMD. A sua plataforma permite que a codificação possa ser escrita para API CUDA tanto quanto para outras linguagens como o C/C++. Para esta linguagem são necessários ajustes em comandos específicos nesta plataforma. Comumente os programadores com alguma experiência na linguagem C/C++ não encontram dificuldades com estas adaptações (AMD, 2016).

O padrão OpenCL é composto por arquitetura de abstração de baixo nível do hardware, para utilizar este padrão os dispositivos devem obedecer a semântica empregada nesta arquitetura, sendo necessário o mapeamento de suas características físicas e a abstração embutida (AMD, 2016).

3.3.4 – CUDA

A arquitetura CUDA consiste de plataforma de computação paralela, sendo sua composição um modelo de programação criado pela NVIDIA em 2006. O seu objetivo consiste em proporcionar maior benefício para alcançar o desempenho computacional elevado, tirando proveito de uma proporção maior de recursos e de unidades de processamento gráfico (GPU). Em síntese através do processamento da API CUDA ocorre uma integração dos códigos C, C++ e Fortran diretamente à GPU, sendo possível dispensar a necessidade de desenvolvimento de nova linguagem de compilação (LINDHOLM, 2008).

Figura - GPU exposta em uma placa de vídeo

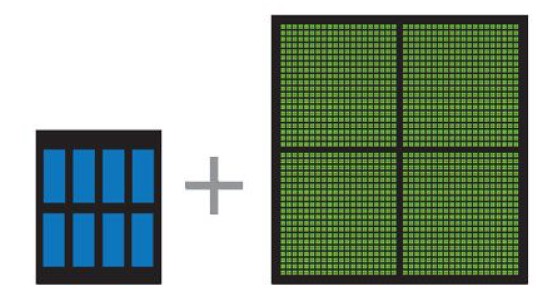


Fonte: (NVIDIA, 2013)

A tecnologia CUDA (p) é de abordagem proprietária, tendo a filosofia de conceder a comunicação direta com o hardware gráfico específico da NVIDIA (r). A Figura 10 demonstra a GPU, que fica encarregada pelo processamento de cada pixel da tela, gerando, a imagem proveniente de um alto número de operações.

A arquitetura de GPU típica e preparada para CUDA. Ela é organizada em uma matriz de multiprocessadores de *streaming* (SMs) altamente encadeados. SMs formam um bloco em que o número de SMs presentes no bloco pode variar de uma geração de GPUs CUDA para outra geração. Cada SM tem um certo número de processadores de *streaming* (SPs) que compartilham a lógica de controle e a cache de instruções. Cada GPU tem DRAM GDDR (*Graphics Double Data Rate*) chamada de memória global. Elas diferem das DRAMs do sistema de placa mãe CPU porque são basicamente a memória do frame buffer que são usadas pelos gráficos. No cálculo elas funcionam como uma memória fora do chip com largura de banda muito alta. Para aplicações maciçamente paralelas a largura de banda mais alta pode gerar maior latência (CUDA, 2007).

Figura 11 - As placas de vídeo têm milhares de núcleos para processar cargas em paralelo



|  |  |
| --- | --- |
| **CPU (múltiplos núcleos)** | **GPU (milhares de núcleos)** |

Fonte: (NVIDIA, 2008)

A Figura 11 demonstra a diferença de arquitetura da CPU com múltiplos núcleos (em azul) e o da GPU com milhares núcleos (em verde).

Na composição da arquitetura CUDA ainda é incluso 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. O NVIDIA pretendia que a nova família de processadores gráficos fosse usada para computação de uso geral (MIVULE, 2008).

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 em geral e não específico, para gráficos com poder de processamento paralelo devido as motivações de seu desenvolvimento (BAKHODA, 2009).

As unidades de execução na GPU ganharam acesso arbitrário de leitura e gravação a memória e acesso a um *cache* gerenciado por softwares, conhecido como memória compartilhada (CUDA, 2020). Todos esses recursos da arquitetura CUDA foram adicionados a GPU para ganhar desempenho em cálculo e executar bem as tarefas gráficas tradicionais (RUETSCH, 2009).

Quando comparada ao *pipeline* de um processador de dados de uma unidade central de processamento (CPU tradicional), a execução de cálculos de uso geral em uma GPU é um novo conceito (AUGUST, 2005).

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 (LINDHOLM, 2008).

A CPU possui uma arquitetura composta por grandes memórias *cache* as quais são fornecidas para reduzir as latências de acesso a instruções e dados de aplicações grandes e complexas. A Lógica de controle nem as memórias *cache* contribuem para a velocidade máxima de cálculo. Os processadores de uso geral precisam satisfazer os requisitos dos sistemas operacionais, aplicações e *devices* de E/S, o que torna a largura de banda da memória mais difícil de aumentar. Com os modelos de memória mais simples e com menos restrições associadas, os projetistas das GPUs podem alcançar uma largura de banda de memória mais alta com facilidade, e os *chips* tem maior uso para os cálculos de ponto flutuante (LINDHOLM, 2008).

Um desafio que ainda permanece e que as unidades aritméticas de ponto flutuante das GPUs são principalmente de precisão simples, porém isso está mudando com as GPUs recentes cuja velocidade de execução em precisão dupla aproxima-se da metade daquela de precisão simples e assim cada vez mais tornando-as adequadas para mais aplicações numéricas (LINDHOLM, 2008).

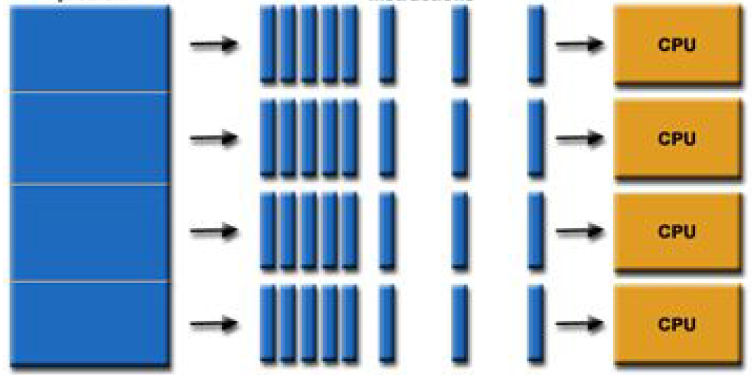
CUDA foi especialmente criada para aproveitar o mecanismo de processamento paralelo das GPUs NVIDIA a fim resolver problemas computacionais complexos. Para programar segundo a arquitetura CUDA™, os desenvolvedores hoje em dia podem usar a linguagem C (SANDERS, 2010). O desenvolvimento de várias técnicas de simulação para análise de desempenho é necessário para projetar uma arquitetura de software extremamente flexível e extensível. Para isso a utilização de vários padrões de projeto de software é crucial para a efetivação de tais requisitos (GAMMA, 1995).

Para o desenvolvimento de sistema utilizando a API CUDA da Nvidia é necessário observar alguns aspectos do problema a ser resolvido, tais como se o problema e paralelizável (AMDAHL, 1967). É necessário possuir conhecimento prévio da arquitetura física da placa na qual se deseja programar, para obter maior benefício da arquitetura CUDA (MIVULE, 2008).

A programação paralela consiste em executar simultaneamente várias partes de uma mesma aplicação, sendo possível a partir do desenvolvimento de sistemas operacionais multi-tarefa, *multi-thread* e paralelos. As aplicações são executadas paralelamente em um mesmo processador, em uma máquina com um multiprocessador ou em um grupo de máquinas interligadas que se comportam como uma só máquina visando o ganho de velocidade. Os *devices* CUDA aceleram a execução dessas aplicações reunindo grande quantidade de paralelismo de dados (SANDERS, 2010).

O paralelismo de dados refere-se a propriedade do programa em que muitas operações aritméticas podem ser seguramente realizadas, de uma maneira simultânea, como na multiplicação de matrizes de grandes dimensões, que podem ter grande quantidade de paralelismo de dados. Executando muitos produtos escalares paralelos em *devices* CUDA é possível acelerar a execução de multiplicação de matrizes, em relação a CPU com processamento tradicional (SANDERS, 2010).

Figura 12 - Divisão de tarefas de um problema entre as CPUs



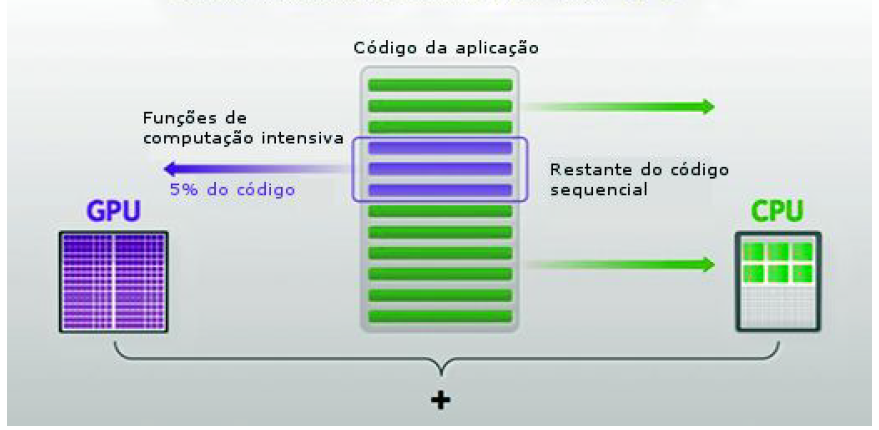
|  |  |  |
| --- | --- | --- |
| **Problema** | **Instruções (t1, t2... tn)** | **Núcleo** |

Fonte: (NVIDIA, 2008).

A Figura 12 demonstra a partição de um problema, que pode ser a implementação de um algoritmo, em quatro tarefas cujas instruções são executadas paralelamente por diferentes CPUs.

Em programação ao utilizar CUDA consiste na execução no *host* (CPU) e em *devices* como a GPU. Sendo que as execuções que não exigem tanto paralelismo são executadas no *host* e as que exigem maior quantidade de paralelismo de dados no GPU. O compilador C da NVIDIA (nvcc) separa os dois durante o processo de compilação. As funções *kernel* normalmente geram um grande número de *threads* para explorar o paralelismo de dados. No cálculo da multiplicação de matriz o cálculo inteiro de multiplicação pode ser implementado como um *kernel,* no qual cada *thread* é utilizado para calcular um elemento da matriz de saída resultante. Tendo como exemplo a composição de uma matriz *1000 x 1000* o *kernel* utiliza a *thread* para realizar os cálculos, gerando 1000000 *threads* quando invocado. As *threads* em CUDA usam poucos ciclos para serem geradas e escalonadas devido o suporte eficiente do *hardware*, que e o contrário do que ocorre com CPUs, que normalmente exigem milhares de ciclos de *clocks* para serem geradas e escalonadas. Assim tirando melhor proveito do paralelismo de dados. Todas as *threads* que sãoexecutadas durante uma chamada são conhecidas coletivamente como *grid* (SANDERS, 2010).

Figura 13 - Aceleração de código utilizando GPU



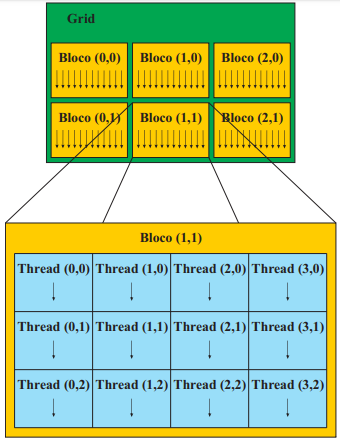
Fonte: (NVIDIA, 2008).

A Figura 13 demonstra parte do código de uma aplicação executada de forma paralela em GPU, um laço de repetição com iterações definidas. Embora a porcentagem de código executada em GPU seja pequena, o aumento de performance é significativo comparado ao desempenho do código executado inteiramente de forma sequencial.

## 3.4 – Solução do sistema linear em GPU CUDA

A solução do sistema linear da Equação (26) pode ser resolvida usando recursos GPU CUDA, com processamento paralelo através da distribuição defluxos de processamentos independentes *(threads)* que são características da arquitetura.

Figura 14 - Elementos da arquitetura CUDA



Fonte: NVIDIA CUDA (NVIDIA, 2013).

Na Figura 14 são demonstrados os principais elementos da arquitetura CUDA. O processo de distribuição dos *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 processados simultaneamente em memória compartilhada.

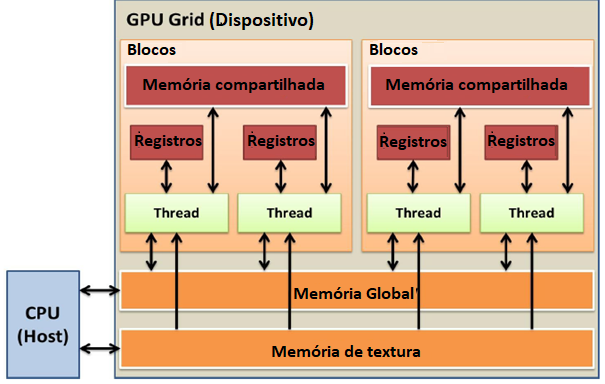
O processo de comunicação entre *blocks* ocorre através da memória global, que possui desempenho inferior a memória compartilhada. A identificação de *blocks* e *threads* é composta de identificadores tridimensionais, onde através destes o desenvolvedor tem acesso a *thread* a qual deve ocorrer o processamento (SANDERS, 2010).

Existe assim uma independência dos núcleos de processadores, possibilitando múltiplas instruções e admitindo o *hyperthreading* com vários *threads* de hardware, que são projetados para maximizar a velocidade de execução dos programas sequenciais (SUTTER, 2005).

A comunicação com a memória global tem desempenho inferior a memória compartilhada, assim, é importante o uso de grandes memórias de *cache* para reduzir a latência de acesso a instruções e dados de aplicações grandes e complexas, já que tanto a lógica de controle quanto as memórias *cache* não contribuem para a velocidade máxima de cálculo (SANDERS, 2010).

A largura de banda da memória é outra questão importante, para minimizar o acesso à memória dinâmica principal (DRAM). Os processadores de uso geral precisam satisfazer os requisitos do sistema operacional, aplicações e *devices* de E/S, e em decorrência deste fato existe uma maior dificuldade de aumentar a capacidade da largura de banda (NVIDIA, 2013).

Figura 15 - Hierarquia de funcionamento de memória em GPU



Fonte: (NVIDIA, 2013).

Na Figura 15 é demonstrado como a hierarquia de memória em GPU funciona. A programação paralela de alto desempenho na maior parte dos chips exigira algum conhecimento de como o hardware realmente funciona, no modelo de programação CUDA o foco é no paralelismo de dados, no alto desempenho e na alta confiabilidade em suas aplicações. Assim as GPUs programáveis modernas são em sua essência composta por *multithreading* e memórias *cache,* relativamente pequenas em comparação com as CPUs. O projeto de interface de memória é centrado na largura de banda.

3.4.1 – CUDA *Library*

Conjunto de ferramentas e produtos CUDA distribuídos com bibliotecas que oferecem novas e ampliadas funcionalidades, correções de bugs e melhorias de desempenho para ambientes com uma ou várias GPUs (NVIDIA, 2019).

A Transformação rápida de Fourier (*Fast Fourier Transform*) é um dos algoritmos numéricos mais importantes e amplamente utilizados na física computacional e no processamento geral de sinais. O processo consiste em dividir para conquistar, propiciando cálculos eficientes do conjunto de dados complexos (AZ, 2007).

O produto da NVIDIA® CUDA ™ *Fast Fourier Transform* (FFT) é composto por duas bibliotecas separadas o cuFFT e cuFFTW. A cuFFT é uma biblioteca que fornece uma interface simples para utilizar FFTs em GPUs da NVIDIA, permitindo aos usuários rapidamente utilizar cálculos de ponto flutuante e paralelismo na GPU. A biblioteca cuFFTW é fornecida como uma ferramenta de transferência, para permitir que os usuários de FFT, baseados nas CPUs mais populares e eficientes comecem a usar as GPUs com um esforço mínimo (NVIDIA, 2019).

3.4.2 – cuBLAS *Library*

A biblioteca cuBLAS é uma implementação do BLAS (*Basic Linear Algebra Subprograms*), ou seja, são subprogramas básicos de álgebra linear, relacionados com o tempo de execução da NVIDIA®CUDA™. A biblioteca permite ao usuário acessar recursos das GPUs da NVIDIA (NVIDIA, 2019).

A Biblioteca cuBLAS é composta por três conjuntos de API:

1) A cuBLAS API (a partir do CUDA 6.0);

2) O CUBLASXT API (a partir do CUDA 6.0);

3) O cuBLASLt API (a partir do CUDA 10.1).

Para utilizar o cuBLAS API, a aplicação deve alocar as matrizes e os vetores necessários no espaço de memória da GPU, preenchê-los com dados, chamar a sequência de cuBLAS desejada e depois carregar os resultados do espaço de memória da GPU de volta para o host. A API cuBLAS também fornece funções para ajudar na escrita e recuperação de dados a partir da GPU.

Para utilizar a API CUBLASXT, a aplicação deve manter os dados no *host* e a biblioteca ira se encarregar de despachar a operação para uma ou várias GPUs presentes no sistema, dependendo da solicitação do usuário.

O cuBLASLt é uma biblioteca leve dedicada a operações de multiplicação em geral de matrizes (GEMM), uma API mais flexível. Esta biblioteca adiciona flexibilidade no layout de dados das matrizes, tipos de entrada, tipos de cálculos, e também na escolha do algoritmo de implementação e heurística através da programação de parâmetros. Após um conjunto de opções para a operação de GEMM pretendida serem identificadas pelo utilizador, estas opções podem ser usadas repetidamente por diferentes entradas. Isto é análogo a como cuFFT e FFTW trabalham, ou seja, primeiro criam um plano e o reutilizam para o mesmo tamanho e tipo de FFTs com dados de entrada diferentes.

3.4.3 – cuSparse *Library*

A biblioteca cuSparsecontém um conjunto de sub-rotinas lineares básicas de álgebra utilizadas para manusear matrizes esparsas. Ela é implementada em cima do NVIDIA® CUDA™ *runtime* (que faz parte do CUDA Toolkit) e foi concebida para ser chamada a partir de C e C++. As rotinas da biblioteca podem ser classificadas em quatro níveis de categoria.

Nível 1 operações entre um vetor em formato esparso e um vetor em formato denso. Nível 2 operações entre uma matriz em formato esparso e um vetor em formato denso. Nível 3 operações entre uma matriz em formato esparso e um conjunto de vetores em formato denso. Nível 4 operações que permitem a conversão entre diferentes formatos de matriz.

A cuSparse permite ao usuário acessar os recursos computacionais da GPU, mas não permite paralelismo entre várias GPUs.

3.4.4 – cuSolver *Library*

A biblioteca cuSolver da NVIDIA fornece uma coleção de solucionadores diretos densos e esparsos que fornecem aceleração significativa para aplicativos de Visão Computacional, CFD, Química Computacional e Otimização Linear (NVIDIA, 2019).

O cuSolver é um pacote de alto nível baseado nas bibliotecas cuBLAS e cuSPARSE. Consistem em dois módulos de APIs. A primeira API cuSolver é utilizada em apenas uma única GPU. A segunda API cuSolverMGé utilizada em nó único para multiGPUs. Cada uma delas pode ser utilizada independentemente ou em conjunto com outras bibliotecas. Para simplificar a notação, cuSolver indica API de GPU única e cuSolverMgindica API multiGPU (NVIDIA *cuSOLVER Library*, 2019).

O cuSolver combina três componentens separados para trabalhar diretamente com as matrizes. O cuSolverDn para rotinas de matriz densa, cuSolverSp para rotinas de matriz esparsa, cuSolverRf para re-fatoração esparsa (útil para resolver uma sequência de matrizes em que apenas os coeficientes são alterados, mas a esparsidade padrão permanece) (NVIDIA, 2019).

O cuSolverDn trabalha com fatoração LU, QR e Cholesky que são solucionadores lineares baseados em decomposições LU, QR e Cholesky. Autovalor simétrico e solucionador de problemas de valor singular. Decomposição de valor singular. Todos os subprogramas têm quatro versões correspondentes a quatro tipos de dados:

S – float (precisão única real),

D – dupla (precisão dupla real),

C – precisão única complexa,

Z – precisão dupla complexa.

OcuSolverSpfornece um conjunto de rotinas esparsas com base em fatoração QR esparsa. Nem todas as matrizes têm bom padrão de esparsidade para paralelismo em fatoração, assim, a biblioteca cuSolverSP possibilita que as CPUs possam trabalhar com matrizes sequenciais. Para aquelas matrizes com paralelismo abundante, o uso da GPU proporcionará maior desempenho. A biblioteca foi projetada para ser chamada de C e C ++.

OcuSolverRf é um pacote de re-fatoração esparsa, que pode fornecer bom desempenho ao resolver uma sequência de matrizes em que apenas os coeficientes são alterados, mas o padrão de escarsidade permanece o mesmo.

Por padrão ao utilizar a GPU da biblioteca cuSolveré assumido que os dados já estão na memória do dispositivo. É responsabilidade do desenvolvedor alocar memória e copiar dados entre a memória da GPU e a memória da CPU usando as rotinas padrões da API CUDA, tais como *cudaMalloc(), cudaFree(), cudaMemcpy()* e *cudaMemcpyAsync().*

3.4.5 – Ladrilhamento

Com o objetivo de alcançar a eﬁciência máxima que a GPU pode propiciar é apresentada uma outra abordagem de multiplicação de matrizes, também utilizando GPU. O uso da memória compartilhada é o acesso a cada bloco de *threads,* objetivando a redução da quantidade gerada de acessos a memória global do dispositivo (LI, 2019).

Figura 16 - Demonstração da multiplicação de matrizes por ladrilhamento



Fonte: Adaptado de LI (LI, 2019)

O procedimento consiste que as matrizes sejam subdivididas em pequenos blocos como demonstrado na Figura 16. Tal técnica é reconhecida como multiplicação por ladrilhamento e em contraste com a abordagem anterior, a qual toda linha de *A* e coluna *B* eram multiplicadas de uma única vez produzindo assim um elemento de *C*, no ladrilhamento tem-se submatrizes de *A* e *B* que geram valores parciais de um elemento de *C*. Assim que todos os ladrilhos forem processados o valor ﬁnal de cada um dos elementos da matriz *C* será o somatório de cada elemento parcial obtido pela multiplicação de cada uma das linhas e colunas das submatrizes *A* e *B*, respectivamente.

## 3.5 – Gêmeo Digital

O conceito foi originalmente introduzido em 2003 por Michael Grieves e divulgado pela NASA pela primeira vez em 2012. Os gêmeos digitais integram todas as informações (eletrônicas) e conhecimentos gerados durante a vida útil de um produto, desde a definição e concepção do produto até o final de sua vida útil. Eles unem o mundo virtual e o mundo real com o objetivo de modelar, entender, prever e otimizar seus ativos reais correspondentes (HARTMANN, 2020).

Saddik (EL SADDIK, 2018) define gêmeo digital com uma réplica digital de uma entidade física viva ou não viva. Ao unir o mundo físico e o virtual, os dados são transmitidos sem interrupções, permitindo que a entidade virtual exista simultaneamente com a entidade física. Gêmeos digitais facilitam os meios para monitorar, entender e otimizar as funções de todas as entidades físicas e, para os seres humanos, fornecem feedback contínuo para melhorar a qualidade de vida e o bem-estar (EL SADDIK, 2018).

Para que os gêmeos digitais possam unir o mundo virtual e o mundo real faz-se necessário avanços exponenciais de várias tecnologias importantes, como hardware de computação, algoritmos matemáticos, gráficos de conhecimento e tecnologias semânticas, além de dispositivo de realidade virtual (SCHALLER, 1997).

O desenvolvimento da tecnologia virtual e de aquisição de dados tem elevado o DT, possibilitando gradualmente que está possa ser uma das principais direções de pesquisa de manufatura inteligente. Sua camada de processamento de informações é composta pelo armazenamento de dados, processamento de dados e mapeamento de dados. Em virtude de ampla gama de processamento de dados eles tendem a necessitar de paralelismo de dados para a efetiva utilização do DT (ZHENG, 2018).

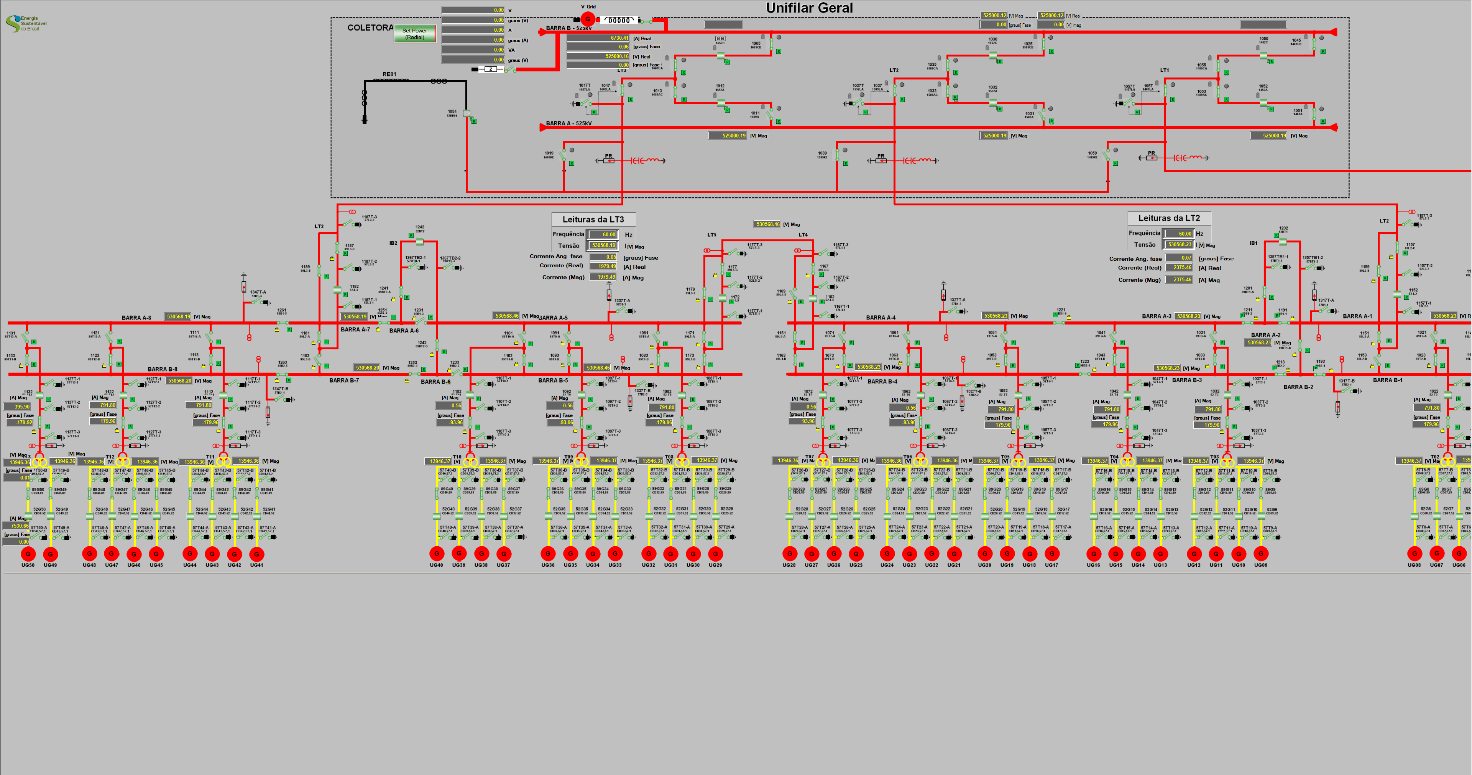
Um crescimento contínuo da capacidade das arquiteturas de computadores e hardware, além do dimensionamento do desempenho dos chips e das GPUs (ARDEN, 2010). Referente à lei de *More-than-Moore*, na época um grande avanço cientifico para a matemática e informática. Isso levou a um crescimento exponencial da capacidade, contribuindo significativamente para a eficiência, escalabilidade e usabilidade de abordagens baseadas em modelos (simulação) e baseadas em dados (RÜDE, 2018).

Com a crescente demanda por melhor desempenho na atualização das medidas de tensão, corrente e potências, simuladas no circuito elétrico, é imprescindível maior confiabilidade das ações e supervisão de equipamentos da rede elétrica. Assim é importante o ganho computacional e assertividade em geral, reduzindo as indisponibilidades na geração de cálculos, sejam elas de natureza acidental ou programada. A simulação complexa dos modelos que envolvem o desenvolvimento de um DT (KRITZINGER, 2018). Resultando em algoritmos de alto custo computacional e dessa forma se faz necessário a redução do tempo computacional gastos pelo sistema, para resolução de sistemas ou análise de circuitos.

3.5.1 – Aplicação na usina hidroelétrica de Jirau

A agilidade em um sistema é crucial, sendo que a aceleração e sua independência de processos pode permitir ganhos de eficiência, principalmente na implantação de sistemas complexos que necessitem constantemente ser reconfigurados. Uma limitação no uso da metodologia é a impossibilidade de comunicação em tempo de execução no envio de dados entre o gêmeo digital e o simulador para o robô, tais processo podem ser paralelizados e com isso o ganho computacional viabiliza a utilização do gêmeo digital.

Figura 17 - Planta digital de funcionando da Usina Hidrelétrica de Jirau

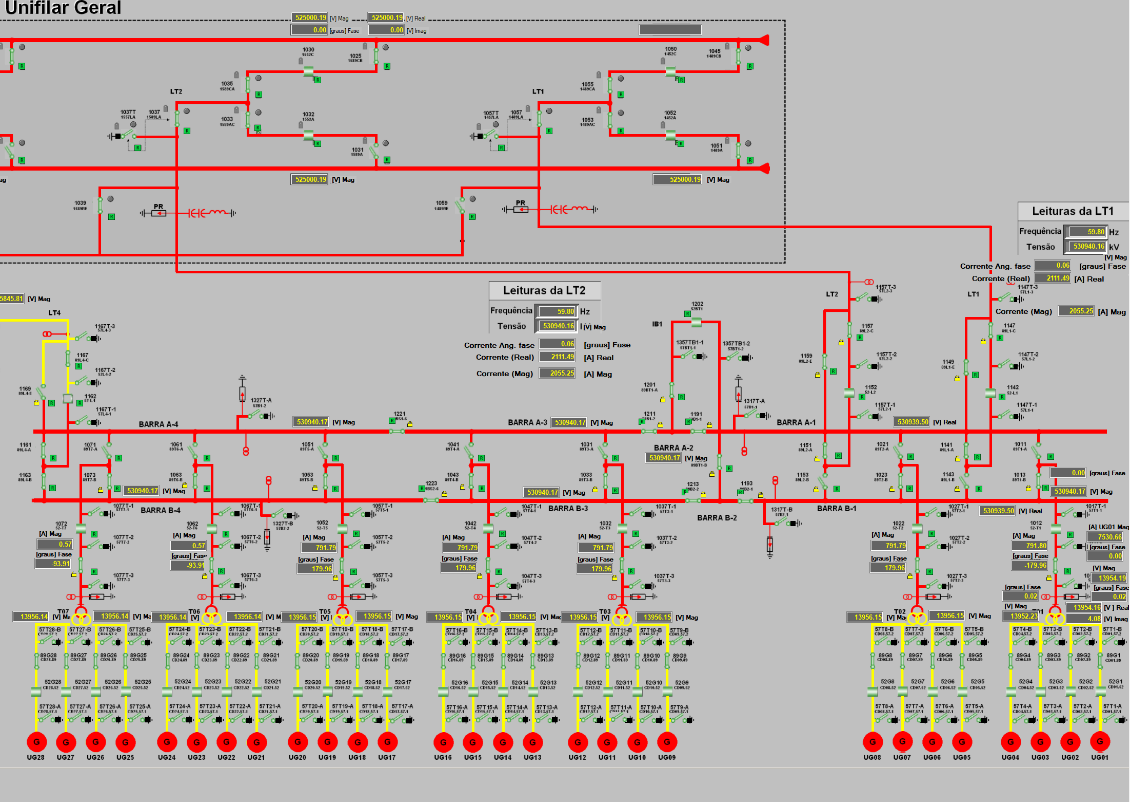
****

Fonte: Usina Jirau (JIRAU, 2020)

Na Figura 17 é demonstrado a planta digital de funcionando da Usina Hidrelétrica de Jirau. O projeto caracteriza-se pela disposição de duas casas de força. Sendo uma nova fronteira tecnológica de turbinas e geradores, necessitando de maior segurança energética para o país.

Cada uma das casas de força apresenta duas áreas equipadas para montagem e manutenção das 50 unidades geradoras, com 75 MW de potência unitária. A barragem principal, do tipo enrocamento com núcleo asfáltico, está disposta segundo um eixo retilíneo ligando a extremidade sul da Ilha do Padre à parede direita da Casa de Força da Margem Esquerda.

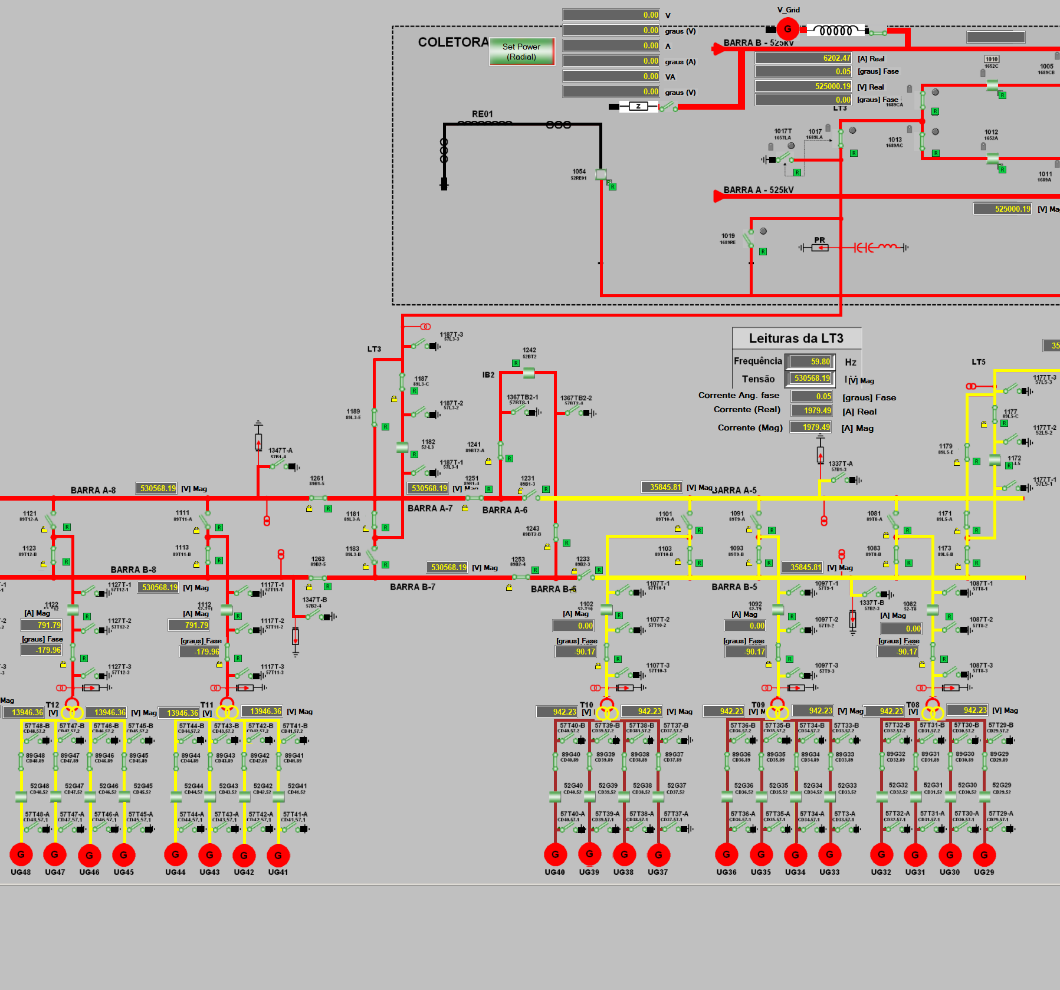
Figura 18 – Lado direito da Usina Hidrelétrica de Jirau



Fonte: Usina Jirau (JIRAU, 2020)

Na Figura 18 é demonstrado o lado direito com 28 unidades geradoras, do tipo bulbo, acoplada à tomada d'água, localizada no braço direito do Rio Madeira.

Figura 19 - Lado esquerdo da Usina Hidrelétrica de Jirau



Fonte: Usina Jirau (JIRAU, 2020)

Na Figura 19 é demonstrado a margem esquerda, localizam-se mais 22 unidades geradoras, também do tipo bulbo, tendo como vértice a extremidade sul da Ilha do Padre.

Para a resolução de sistemas lineares e resolução de circuitos elétricos do GD e necessário obter ganho computacional nas aplicações da usina hidroelétrica de Jirau, e explorar o uso gêmeo digital em simulações e modelagem, utilizando os recursos disponíveis na GPU e empregando a arquitetura CUDA para paralelização de algumas etapas envolvidas na simulação e paralelização de processos do Gêmeo digital.

Com o intuito de obter o máximo de proveito dos processadores no cenário atual e importante que os *softwares* possam ser projetados para executar processos em paralelo. Com essa premissa temos uma heterogeneidade a qual pode proporcionar que o processo de desenvolvimento de software apresente um número maior de ferramentas e arquiteturas a serem utilizadas. Assim possibilitando a ocorrência de uma série de desafios para a comunidade de programação, como resultado uma gama maior de possibilidades usando CUDA (NVIDIA, 2016).

Em tese para avaliar o desempenho de cada API estamos realizando execuções de algoritmos e os seus tempos decorridos nestas execuções foram computados. Como métrica de avalição tem se avaliado a média como o valor utilizado e através deste tempo de execução obteve-se um intervalo de confiança das amostras. Consequentemente é observado quando ocorre um *speedup,* obtido em cada algoritmo após a execução de cada API.

## 3.6 – Considerações do Capítulo

Neste capítulo foi comentado sobre a computação paralela e arquitetura CUDA. Iniciando sobre a noção de aceleração. Comentado sobre a programação paralela e multi-processamento aberto OpenMP, OpenACC e OpenCL. Apresentado abordagem por ladrilhamento para alcançar eﬁciência máxima de uso da GPU. Explicado sobre o uso para o gêmeo digital.

# Capítulo 4 – Materiais e Métodos

Este capítulo descreve como foi o processo para alcançar um melhor desempenho para a resolução do sistema linear através do uso de GPU. Aqui serão estabelecidas as estratégias para alcançar o objetivo final, juntamente com a escolha e especificação de uma série de ferramentas que irão auxiliar no processo.

## 4.1 – Método cientifico adotado

O código para a resolução do sistema linear, inicialmente não foi desenvolvido pensando em ser executado com o uso de GPU, as estruturas de dados utilizadas, a movimentação de dados e o fluxo do programa em si não estavam adequados para tal processamento. A resolução de sistema linear é um processo que tem elevado custo computacional, onde o tempo é diretamente proporcional ao tamanho da matriz.

Assim foi adotado o processo de desenvolvimento que possa guiar a tarefa de paralelização, do código existente, para torná-la menos difícil, porém não menos desafiadora. A ideia do processo é identificar inicialmente uma parte de código que possa ser acelerada por GPU, desenvolver uma versão de código paralelo inicial, aplicar otimizações para gerar melhoria no código, isto antes de implantar uma nova versão do código e retornar ao início do processo novamente enquanto melhorias ainda forem possíveis. A seguir os passos para a resolução do sistema linear são discutidos em mais detalhes.

4.1.1 – Avaliar e analisar métodos diretos e iterativos

Comparar os processos pelos algoritmos SVD, Gauss, QR, LU, Cholesky e por Inversão da Matriz. Avaliar e analisar a execução da aplicação em um cenário com dados reais e consistentes, ao coletar os dados estes são utilizados para identificar partes do programa que mais contribuem no tempo total da resolução. Uma análise para estimar o valor de *speedup* é analisada de acordo com as leis de Amdahl e Gustafson. As ferramentas que foram utilizadas neste trabalho são descritas na seção 4.4.

Os dados reais foram obtidos pelo gerador de matrizes para MNA da Usina Hidrelétrica de Jirau. O programa gera matrizes *A, x* e *b* do sistema linear Ax = b, com a origem a partir de uma *netlist*. Obtém-se as matrizes geradas pelo arquivo *“uh\_jirau.ckt”* que representa o circuito elétrico para a formatação da leitura em CUDA. A estrutura de saída do arquivo é formada pelo primeiro elemento sendo a linha da matriz, o segundo elemento é a coluna da matriz e o terceiro elemento é o valor da matriz. Os valores são limitados a 2 casas decimais.

Para todos os métodos diretos e indiretos na CPU e GPU foram utilizadas inicialmente a leitura das matrizes do circuito elétrico, através da utilização do Gerador de Matrizes para *Modified Nodal Analysis*. O tempo deste processo não foi computado para as análises.

Para executar o programa é preciso criar uma pasta com o nome “*files”* no mesmo diretório do executável “MNA.exe”, esta pasta precisa necessariamente estar vazia antes da execução do programa ou poderá resultar em falhas. O arquivo *ckt* que será analisado também deve estar no mesmo diretório do executável e com o nome “netlist.ckt”.

Após a execução do programa serão gerados 2 arquivos:

- "/Files/MatrixA.mtx" |

|- "/Files/ElapsedTime.txt"

Onde como exemplo de saída do primeiro arquivo teremos dimensões como <3573> <3573> <13482>. O segundo arquivo registra o tempo de geração do circuito, este não foi levado em consideração nos experimentos.

4.1.2 – Avaliar o desempenho dos experimentos executados na CPU

Avaliar o desempenho dos experimentos executados na CPU, são processados 100 vezes para os algoritmos SVD, Gauss, QR, LU e Cholesky. A Inversão da matriz também é utilizada. É utilizada a programação paralelizada por OpenMP.

Nesta etapa são utilizadas bibliotecas por diretivas para o pré-processador indicando áreas de código paralelizáveis ou por meio da programação através de uma API para programação paralela do OpenMP. Entretanto, para alguns casos é necessário realizar refatorações no código sequencial para que seja possível criar uma versão de código paralelo.

4.1.3 – Avaliar o desempenho dos experimentos executados na GPU

Avaliar o desempenho dos experimentos executados na GPU, são processados 100 vezes para os algoritmos SVD, Gauss, QR, LU e Cholesky. A Inversão da matriz também é utilizada. É utilizada a programação paralelizada por CUDA.

Paralelizar depois de identificar os pontos do programa onde se concentra a maior parte do tempo de execução, é realizado o estudo para a paralelização do código. Nesta etapa são utilizadas bibliotecas aceleradas por GPU, diretivas para o pré-processador indicando áreas de código paralelizáveis ou por meio da programação de GPU, através de uma API para programação paralela CUDA.

4.1.4 – Comparar os experimentos executados na CPU x GPU

Comparar pela mediana o desempenho dos experimentos executados na CPU e GPU, são comparados para avaliar os 3 (três) melhores algoritmos utilizados em OpenMP e CUDA.

Verificar a possibilidade de otimizar as etapas, após a paralelização, para melhorar o desempenho obtido. As otimizações podem ser implementadas aos poucos e de forma incremental, devido a característica iterativa do processo. Dentre as otimizações possíveis, podemos destacar a gestão eficiente de memória, considerando estratégias como a sobreposição das tarefas de transferência e execução, o uso de memória compartilhada na GPU e estratégias para otimizar o acesso à memória global do dispositivo.

4.1.5 – Analisar o custo computacional dos algoritmos

O custo computacional dos algoritmos é importante ser levado em consideração, para a avaliação dos recursos necessários e destinados para o experimento.

Uma versão de código gerado durante o processo é uma etapa bastante importante. Ela pode ser feita mesmo que através de otimizações e implementações parciais, reduzindo os riscos que grandes mudanças podem trazer. Após uma versão ter sido implantada o processo entra em um novo ciclo, passando novamente por todas as etapas.

4.1.6 – Comparar a paralelização em diferentes GPUs

Para o desenvolvimento das etapas anteriores exploramos o uso de bibliotecas aceleradas por GPU. Sendo necessário a execução em diferentes GPUs para garantir a portabilidade do código, como também verificação de ganho computacional em diferentes *devices*.

O uso de bibliotecas pode facilitar a tarefa de paralelização e otimização de código, o que é feito com relativamente pouco esforço, quando comparado com o trabalho envolvido na implementação específica de uma solução.

## 4.2 – O processo de investigação com técnicas, bibliotecas e ferramentas

Para o trabalho proposto são consideradas as seguintes bibliotecas de acordo com as suas possíveis contribuições para o trabalho:

**CUBLAS** é uma biblioteca que possibilita o uso de recursos básicos de álgebra linear. As bibliotecas *cuFFT* e *cuFFTW* possibilitam cálculos eficientes em conjunto de dados complexos permitindo maior eficiência.

**CUSOLVER** é uma biblioteca baseada nas bibliotecas CUSPARSE e CUBLAS. Esta reúne diversos métodos acelerados por GPU, para a solução de sistemas de equações lineares tanto densos quanto esparsos. A biblioteca oferece alguns métodos diretos e iterativos para obter a solução dos sistemas de equações, além de oferecer rotinas de reordenação e pré-condicionamento destes sistemas.

A biblioteca possibilita otimização linear, permitindo a resolução do sistema linear para os métodos diretos LU, Cholesky ou QR. Os métodos iterativos como Gauss Jacobi, Gauss Seidel ou SVD, são baseados na divisão da multiplicação da matriz e a solução triangular esparsa, implementada na biblioteca *cuSPARSE e cuBLAS* na GPU.

**MAGMA** é uma biblioteca MAGMA (*Matrix Algebra for GPU and Multicore Architectures*) tem como objetivo fornecer uma biblioteca de álgebra linear, com algoritmos que possibilitem uma solução precisa no menor tempo possível.

Para atingir este objetivo, a biblioteca explora as arquiteturas heterogêneas, oferecendo algoritmos paralelizados tanto em CPU quanto em GPU. Oferece métodos e rotinas semelhantes aos que a biblioteca CUSPARSE, CUBLAS e CUSOLVER.

**Thrust** esta biblioteca é baseada na biblioteca *Standard Template Library* (STL) da linguagem C++. Fornece mais flexibilidade e produtividade para a programação em GPU, a biblioteca oferece um conjunto de algoritmos paralelos, além de simplificar a gestão de memória e as transferências entre *host* e *device* no ambiente de programação CUDA.

## 4.3 – Experimentos

Para iniciar a implementação utilizando GPU utilizamos uma função para executar o *Kernel*. Tal função será executada diretamente na GPU e utilizará os índices das *threads* para tratar os dados. Definindo a quantidade necessária de *grids*, blocos e a quantidade de *threads* que serão executados na GPU. Esses parâmetros estão interligados a arquitetura da placa, sendo assim, a familiaridade e o conhecimento do programador sendo mais amplo, tem-se um proveito maior da plataforma.

O processo é o envio dos dados da memória da CPU para serem executados na memória global da GPU. Ao final do processamento interno de execução na GPU os dados fazem o caminho inverso sendo devolvidos a CPU.

Foi implementado algoritmo utilizando a linguagem C++, onde utilizou-se inicialmente apenas o processamento de CPU, posteriormente foi implementado o algoritmo utilizando a GPU, através do acesso a API CUDA.

Para processar os experimentos, a execução ocorreu em equipamento que é composto por máquina que possui sistema operacional Linux, distribuição Ubuntu, com processador intel core i7, 2.5 GHz, 2 núcleos, 3 MB de memória cache e 6GB de memória RAM e uma placa gráfica NVIDIA Geforce 940M de capacidade computacional 5.0 (GPU A).

Posteriormente para a GPU B, foram utilizadas as mesmas configurações da GPU A, porém com a placa gráfica NVIDIA Geforce GTX 1070, de capacidade computacional 6.1.

Para a GPU C é composta pelo sistema operacional Linux, distribuição Ubuntu 2 x Intel Xeon Gold 6140 18 Core 2.30ghz/24.75mb/fclga3647, 2x Placa Nvidia Quadro P5000 16gb Gddr5x 256 Bits, 256gb RAM, 12 cores 24 threads cada processador.

* O procedimento será realizado levando em consideração a implementação pela técnica de ladrilho sequencial, com processos paralelizados nos métodos OpenMP e CUDA. Para a utilização paralelizada com uso em GPU.
* A resolução do sistema linear é executada em cem vezes, com o objetivo de extrair a mediana. Sendo paralelizadas utilizando o OpenMP e CUDA.
* Durante a execução deste processo temos as matrizes e os ladrilhos sendo ajustados para alcançar o máximo desempenho.
* Com a medição do desempenho sendo processada, teremos a verificação do tempo de execução e a *speedup* através da lei de Amdahl (AMDAHL, 1967), e em GFLOPS.

## 4.4 – O processamento dos dados

O conjunto de ferramentas foi escolhido de acordo com algumas funcionalidades oferecidas, como o *profiling* de aplicações, a depuração de código tanto em CPU quanto em GPU, a gerência de memória e verificação da corretude dos resultados numéricos obtidos.

- **gprof e nvprof** estas duas ferramentas são importantes para a fase de avaliação da aplicação em um cenário de teste real. Com elas é possível coletar os dados da aplicação, durante a sua execução, como o número de chamadas de função, tempo de execução, dados sobre a execução de *kernels* e informações sobre transferências de memória entre *host* e *device*.

- **cuda-gdb** provê a funcionalidade de depuração de código, para o tratamento e identificação de erros de programação que podem ocorrer. Ela permite depurar o código tanto em CPU quanto em GPU, possibilitando ainda a visualização das variáveis identificadoras da hierarquia de *threads* CUDA.

- **cuda-memcheck** esta ferramenta permite encontrar erros relacionados a memória da GPU, como acessos fora do limite de uma variável, nos diferentes níveis da hierarquia de memória e erros de alocação e desalocação de memória, dentre outras funcionalidades.

- **nvcc** é o compilador para os programas escritos utilizando CUDA. Ele trabalha em conjunto com os compiladores de linguagem C ou C++, presentes no ambiente de execução. O nvcc separa o código do *host* e do *device,* sendo o código do *device* compilado pelo nvcc e o do *host* pelos outros compiladores.

- ***Script* para verificação de resultados** mais importante que acelerar o código é manter a consistência dos resultados numéricos, que é o código do programa original gerado. Para realizar esta verificação será implementado um *script* para a verificação e comparação dos resultados, os valores finais gerados pelo programa em cada passo de simulação.

No processo adotado pela programação em C++ foi feita a implementação do Algoritmo 1 (abaixo), para a resolução de sistema linear. Os algoritmos são descriminados nas próximas seções.

|  |  |
| --- | --- |
| Algoritmo 1 – Resolução linear | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | Entrada:  **A** = matriz que está sendo multiplicada  **B** = matriz com os valores de potência  Saída:  **C** = matriz com os resultados dos cálculos  **R** = matriz de resultado  # função para multiplicar uma matriz por um vetor  # função decorada para cuda  Função = multiplicar vetor( A, b ):  **C** = **A** \* **b**  Retornar **C**  # função para multiplicar duas matrizes  Função = multiplicar matriz( **A**, **B** ):  # Para nas colunas da matriz ‘**B**’  Para inicio até fim colunas(b) faça  **R**[:, i] = multiplicar 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 função executada no 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 modificada na programação paralela.

## 4.5 – Análise e interpretação dos resultados através dos algoritmos

A inicialização da biblioteca *CUBLAS* cria um identificador para uma estrutura, que mantém o contexto da biblioteca *CUBLAS.* Ele aloca recursos de hardware no host e no dispositivo, deve ser chamado antes de fazer o acesso a outras bibliotecas.

A inicialização da biblioteca *cuSolverDN* cria um identificador no contexto *cuSolverDN.* O qual deve ser iniciado, antes que qualquer outra função da API *cuSolverDN* seja chamada. Sua função é alocar os recursos de hardware necessários para acessar a GPU.

No passo 1 de 3 do *cuSolverDN* cria-se o inicializador:

|  |  |  |
| --- | --- | --- |
|  | *cusolverStatus\_t*  *cusolverDnCreate(cusolverDnHandle\_t \*handle);* |  |

No passo 2 de 3 do *cuSolverDN,* ocorre uma variação dependendo do tipo de algoritmo adotado. Podendo ser utilizado um dos algoritmos como LU, QR, Cholesky, Gauss Jacobi, Gauss Seidel ou SVD.

Para o método iterativo estacionário Gauss Seidel, baseado na divisão, são utilizadas as bibliotecas *cuSPARSE* e *cuBLAS*. A memória apropriada é alocada e definida para a matriz *A*. O método de Cholesky triangular superior são calculados e estão presentes na memória do dispositivo GPU. No Apêndice B.1 consta tabela com os parâmetros de configuração da biblioteca.

4.5.1 – Eliminação Gaussiana

Passo 1 de 3 do *CUBLAS*, criar o inicializador:

|  |  |  |
| --- | --- | --- |
|  | *cublasStatus\_t*  *cublasCreate(cublasHandle\_t \* handle);* |  |

Para a solução geral do sistema linear através da Eliminação Gaussiana é utilizada a biblioteca *cublasCgemm3m* da API *Cublas.* Esta função executa a multiplicação de matriz complexa, usando o algoritmo de redução de complexidade de Gauss. Essa rotina é suportada apenas em GPUs com recursos de arquitetura iguais 5.0 ou superiores.

Passo 2 de 3 do *CUBLAS*, passar os parâmetros das matrizes para a biblioteca *cublasCgemm3m,* processar a Eliminação Gaussiana em CUDA:

|  |  |  |
| --- | --- | --- |
|  | *cublasZgemm3m(cublasHandle\_t handle,*  *cublasOperation\_t transa, cublasOperation\_t transb,*  *int m, int n, int k,*  *const cuDoubleComplex \*alpha,*  *const cuDoubleComplex \*A, int lda,*  *const cuDoubleComplex \*B, int ldb,*  *const cuDoubleComplex \*beta,*  *cuDoubleComplex \*C, int ldc)* |  |

Passo 3 de 3 do *CUBLAS* serve para finalizar o processo. A função *cublasDestroy* libera recursos de hardware utilizados. Essa função geralmente é a última a ser chamada.

|  |  |  |
| --- | --- | --- |
|  | *cublasStatus\_t*  *cublasDestroy(cublasHandle\_t \*handle)* |  |

A liberação desses recursos é realizada chamando o *cublasDestroy,* o qual implicitamente invoca a *cublasDeviceSynchroniz*e e esta atualiza o status da aplicação. É recomendável minimizar o número de ocorrências *cublasCreate / cublasDestroy.* No Apêndice B.2 consta tabela com os parâmetros de configuração da biblioteca.

4.5.2 – Decomposição LU

Para a solução geral do sistema linear através da decomposição LU, são utilizadas as bibliotecas *cusolverDnSgetrf e cusolverDnSgetrs*. A função *cusolverDnSgetrf* responsável por calcular a matriz geral *m x n*, usando a matriz *A* com pivotação parcial com trocas de linhas*.*

No passo 2 de 3 do *cuSolverDN*, são passados os parâmetros das matrizes para a biblioteca *cusolverDnSgetrf,* processar a decomposição LU com precisão única:

|  |  |  |
| --- | --- | --- |
|  | *cusolverDnSgetrf(cusolverDnHandle\_t handle,*  *int m,*  *int n,*  *float \*A,*  *int lda,*  *float \*Workspace,*  *int \*devIpiv,*  *int \*devInfo );* |  |

Para a solução geral do sistema linear com precisão dupla também são utilizadas as bibliotecas *cusolverDnSgetrf* e *cusolverDnSgetrs*. Para esta precisão no passo 2 de 3 do *cuSolverDN*, são passados os parâmetros das matrizes para a biblioteca *cusolverDnSgetrs* processar a decomposição LU com precisão dupla:

|  |  |  |
| --- | --- | --- |
|  | *cusolverDnSgetrs(cusolverDnHandle\_t handle,*  *cublasOperation\_t trans,*  *int n,*  *int nrhs,*  *const float \*A,*  *int lda,*  *const int \*devIpiv,*  *float \*B,*  *int ldb,*  *int \*devInfo );* |  |

A função *cusolverDnDgetrs* utiliza as matrizes *L* e *U* definidas por *cusolverDnDgetrf*, para resolver com precisão dupla o sistema linear através da seguinte chamada em CUDA:

|  |  |  |
| --- | --- | --- |
|  | */\* Decomposição LU \*/*  *if (pivot\_on){*  *status = cusolverDnDgetrf(*  *cusolverH,*  *m,*  *m,*  *d\_A,*  *lda,*  *d\_work,*  *d\_Ipiv,*  *d\_info);*  *}* |  |

No Apêndice B.3 consta tabela com os parâmetros de configuração da biblioteca.

4.5.3 – Decomposição QR

Para a decomposição QR e verificação da ortogonalidade com precisão única, utilizamos as bibliotecas *cusolverDnSgeqrf* e *cusolverDnSorgqr*. A função *cusolverDnSgeqrf* calcula com precisão única. A função *cusolverDnSorgqr* calcula a matriz ortogonal *Q* usando elementos dos vetores armazenados em *A* e referenciando elementos armazenando em outras posições em *d\_tau*.

No passo 2 de 3 do *cuSolverDN*, são passados os parâmetros das matrizes para as bibliotecas *cusolverDnSgeqrf* e *cusolverDnSorgqr* processarem a decomposição QR com precisão única.

|  |  |  |
| --- | --- | --- |
|  | *cusolverDnSgeqrf(cusolverDnHandle\_t handle,*  *int m,*  *int n,*  *float \*A,*  *int lda,*  *float \*TAU,*  *float \*Workspace,*  *int Lwork,*  *int \*devInfo );* |  |

|  |  |  |
| --- | --- | --- |
|  | *cusolverDnSorgqr(cusolverDnHandle\_t handle,*  *int m,*  *int n,*  *int k,*  *float \*A,*  *int lda,*  *const float \*tau,*  *float \*work,*  *int lwork,*  *int \*devInfo);* |  |

A função para calcular a precisão dupla são compostas pelas bibliotecas *cusolverDnDgeqrf, cusolverDnDorgqr* e *cublasStrsm.* As funções *cusolverDnDgeqrf e cusolverDnDorgqr* são similares as bibliotecas *cusolverDnSgeqrf* e *cusolverDnSorgqr* da precisão única. Assim ao obter um sistema triangular este será resolvido pela função *cublasStrsm.*

No passo 2 de 3 do *cuSolverDN*, são passados os parâmetros das matrizes para as bibliotecas *cusolverDnDgeqrf e cusolverDnDorgqr.* Estes processam a decomposição QR com precisão dupla:

|  |  |  |
| --- | --- | --- |
|  | *cusolverDnDgeqrf(cusolverDnHandle\_t handle,*  *int m,*  *int n,*  *double \*A,*  *int lda,*  *double \*TAU,*  *double \*Workspace,*  *int Lwork,*  *int \*devInfo );* |  |

|  |  |  |
| --- | --- | --- |
|  | *cusolverDnDorgqr\_bufferSize(cusolverDnHandle\_t handle,*  *int m,*  *int n,*  *int k,*  *const double \*A,*  *int lda,*  *const double \*tau,*  *int \*lwork);* |  |

O método da decomposição QR para resolver o sistema linear é através da seguinte chamada em CUDA:

|  |  |  |
| --- | --- | --- |
|  | */\* Decomposição QR \*/*  *cusolver\_status = cusolverDnDgeqrf(cusolverH,*  *m,*  *m,*  *d\_A,*  *lda,*  *d\_tau,*  *d\_work,*  *lwork,*  *devInfo);*  *cudaStat1 = cudaDeviceSynchronize();*  *assert(CUSOLVER\_STATUS\_SUCCESS == cusolver\_status);*  *assert(cudaSuccess == cudaStat1);* |  |

|  |  |  |
| --- | --- | --- |
|  | *// Verificar se QR e bom ou nao*  *cudaStat1 = cudaMemcpy(&info\_gpu, devInfo, sizeof(int), cudaMemcpyDeviceToHost);*  *assert(cudaSuccess == cudaStat1);*  *printf("after geqrf: info\_gpu = %d\n", info\_gpu);*  *assert(0 == info\_gpu);* |  |

No Apêndice B.4 consta tabela com os parâmetros de configuração da biblioteca.

4.5.4 – Decomposição de Cholesky

As funções *cusolverDnSpotrf* e *cusolverDnSpotrs* calculam a decomposição de Cholesky para resolução de sistemas lineares, positivos de precisão única e dupla. Para uma matriz hermitiana *m × m*, apenas a parte inferior ou superior é significativa. O parâmetro de entrada *uplo* indica qual parte da matriz é usada. A função deixaria a outra parte intocada.

Se o parâmetro de entrada *uplo* for *CUBLAS\_FILL\_MODE\_LOWER*, apenas a parte triangular inferior de *A* será processada e substituída pelo fator *L* de Cholesky triangular inferior. Se o parâmetro de entrada *uplo* for *CUBLAS\_FILL\_MODE\_UPPER*, apenas a parte triangular superior de *A* será processada e substituída pelo fator *U* de Cholesky triangular superior.

No passo 2 de 3 do *cuSolverDN*, a função *cusolverDnSpotrf* calcula com precisão única uma matriz simétrica positiva definida por *m* x *m* em *A*:

|  |  |  |
| --- | --- | --- |
|  | *cusolverDnSpotrf(cusolverDnHandle\_t handle,*  *cublasFillMode\_t uplo,*  *int n,*  *float \*A,*  *int lda,*  *float \*Workspace,*  *int Lwork,*  *int \*devInfo );* |  |

Para a precisão dupla no passo 2 de 3 do *cuSolverDN*, a função *cusolverDnSpotrs* calcula uma matriz simétrica positiva definida por *m* x *m* em *A*:

|  |  |  |
| --- | --- | --- |
|  | *cusolverDnSpotrs(cusolverDnHandle\_t handle,*  *cublasFillMode\_t uplo,*  *int n,*  *int nrhs,*  *const float \*A,*  *int lda,*  *float \*B,*  *int ldb,*  *int \*devInfo);* |  |

Para invocar a função pela decomposição de Cholesky para resolver o sistema linear através da seguinte chamada em CUDA:

|  |  |  |
| --- | --- | --- |
|  | */\* Decomposição Cholesky \*/*  *status = cusolverDnDpotrfBatched(*  *handle,*  *uplo,*  *m,*  *d\_Aarray,*  *lda,*  *d\_infoArray,*  *batchSize);*  *cudaStat1 = cudaDeviceSynchronize();*  *assert(CUSOLVER\_STATUS\_SUCCESS == status);*  *assert(cudaSuccess == cudaStat1);*  *cudaStat1 = cudaMemcpy(infoArray, d\_infoArray, sizeof(int)\*batchSize, cudaMemcpyDeviceToHost);*  *cudaStat2 = cudaMemcpy(L0, Aarray[0], sizeof(double) \* lda \* m, cudaMemcpyDeviceToHost);*  *cudaDeviceSynchronize();*  *assert(cudaSuccess == cudaStat1);*  *assert(cudaSuccess == cudaStat2);*  *for(int j = 0 ; j < batchSize ; j++){*  *printf("info[%d] = %d\n", j, infoArray[j]);*  *}*  *assert( 0 == infoArray[0] );* |  |

No Apêndice B.5 consta tabela com os parâmetros de configuração da biblioteca.

4.5.5 – Gauss Jacobi

As funções *cusolverDnSsyevj* e *cusolverDnDsyevj* tem funcionalidade pelo método Jacobi. O paralelismo do método Jacobi na GPU tem melhor desempenho em matrizes de pequenas e médias dimensões, para aproximação até uma certa precisão. Possui dois parâmetros para controlar a precisão. O primeiro parâmetro é a tolerância, a precisão da máquina, onde a função *cusolverDnXsyevjSetTolerance* é utilizada para definir a tolerância a priori. O segundo parâmetro é o número máximo de iterações, a função *cusolverDnXsyevjSetMaxSweeps* é utilizada para definir um limite adequado, sendo o valor inicial 100 utilizado.

A verificação da convergência é feita na função *cusolverDnXsyevjGetResidual* e o número de iterações executadas na função *cusolverDnXsyevjGetSweeps*. O valor residual precisa informar um parâmetro de saída menor que zero. Se a informação for maior que 1, então não converge sob uma dada tolerância e iterações máximas.

No passo 2 de 3 do *cuSolverDN*. As funções *cusolverDnSsyevj* e *cusolverDnDsyevj* são de precisão única e precisão dupla, respectivamente:

|  |  |  |
| --- | --- | --- |
|  | *cusolverDnSsyevj(cusolverDnHandle\_t handle,*  *cusolverEigMode\_t jobz,*  *cublasFillMode\_t uplo,*  *int n,*  *float \*A,*  *int lda,*  *float \*W,*  *float \*work,*  *int lwork,*  *int \*info,*  *syevjInfo\_t params);* |  |
|  | *cusolverDnDsyevj(cusolverDnHandle\_t handle,*  *cusolverEigMode\_t jobz,*  *cublasFillMode\_t uplo,*  *int n,*  *double \*A,*  *int lda,*  *double \*W,*  *double \*work,*  *int lwork,*  *int \*info,*  *syevjInfo\_t params);* |  |

Para invocar a função do método Gauss Jacobi para resolver o sistema linear é através da seguinte chamada em CUDA:

|  |  |  |
| --- | --- | --- |
|  | */\** Gauss Jacobi *\*/*  *cusolverEigType\_t itype = CUSOLVER\_EIG\_TYPE\_1;*  *cusolverEigMode\_t jobz = CUSOLVER\_EIG\_MODE\_VECTOR; .*  *cublasFillMode\_t uplo = CUBLAS\_FILL\_MODE\_LOWER;*  *cusolver\_status = cusolverDnDsyevj(*  *cusolverH,*  *itype,*  *jobz,*  *uplo,*  *m,*  *d\_A,*  *lda,*  *d\_B,*  *lda,*  *d\_W,*  *&lwork);*  *assert (cusolver\_status == CUSOLVER\_STATUS\_SUCCESS);*  *cudaStat1 = cudaMalloc((void\*\*)&d\_work, sizeof(double)\*lwork);*  *assert(cudaSuccess == cudaStat1);* |  |

4.5.6 – Gauss Seidel

A função *cusparseDcsrsv\_solve* resolve o sistema linear por triangulares inferiores e superiores pelo método Seidel:

|  |  |  |
| --- | --- | --- |
|  | *cusparseStatus\_t*  *cusparseDcsrsv\_solve(handle, CUSPARSE\_OPERATION\_TRANSPOSE,*  *n, 1.0, descrpR, valR, csrRowPtrR, csrColIndR,*  *inforRt, r, t);*  *cusparseDcsrsv\_solve(handle, CUSPARSE\_OPERATION\_NON\_TRANSPOSE,*  *n, 1.0, descrpR, valR, csrRowPtrR, csrColIndR,*  *inforR, t, z);* |  |

Para invocar a função do método Gauss Seidel para resolver o sistema linear é através da seguinte chamada em CUDA:

|  |  |  |
| --- | --- | --- |
|  | *cusparseDcsrmv(handle, CUSPARSE\_OPERATION\_NON\_TRANSPOSE,*  *n, n, 1.0, descrA, valA, csrRowPtrA, csrColIndA,*  *p, 0.0, q);* |  |

No Apêndice B.6 consta tabela com os parâmetros de configuração da biblioteca.

4.5.7 – SVD

A função *cusolverDnSsyevd* é utilizada pelo SVD para a resolução de sistemas lineares, com precisão única e precisão dupla. Essa função calcula todos os autovalores e, opcionalmente, autovetores de uma matriz simétrica real *A*.

No passo 2 de 3 do *cuSolverDN*, são passados os parâmetros das matrizes para a biblioteca *cusolverDnSsyevd* processar pelo SVD:

|  |  |  |
| --- | --- | --- |
|  | *cusolverDnSsyevd(cusolverDnHandle\_t handle,*  *cusolverEigMode\_t jobz,*  *cublasFillMode\_t uplo,*  *int n,*  *float \*A,*  *int lda,*  *float \*W,*  *float \*work,*  *int lwork,*  *int \*devInfo);* |  |

O parâmetro pode assumir os valores *CUSOLVER\_EIG\_MODE\_VECTOR* ou *CUSOLVER\_EIG\_MODE\_NOVECTOR* para responder se os vetores próprios são desejados. A matriz simétrica *A* pode ser armazenada na parte inferior (*CUBLAS\_FILL\_MODE\_LOWER*) ou na parte superior (*CUBLAS\_FILL\_MODE\_UPPER*). Se os autovetores forem desejados, na saída A haverá autovetores ortonormais. Os autovalores são armazenados em um matriz *W*.

Para invocar a função de SVD para resolver o sistema linear através da seguinte chamada em CUDA:

|  |  |  |
| --- | --- | --- |
|  | *// SVD*  *signed char jobu = 'A'; // Todas m colunas de U*  *signed char jobvt = 'A'; // Todas as n colunas da VT*  *cusolver\_status = cusolverDnDgesvd (*  *cusolverH,*  *jobu,*  *jobvt,*  *m,*  *n,*  *d\_A,*  *lda,*  *d\_S,*  *d\_U,*  *lda, // ldu*  *d\_VT,*  *lda, // ldvt,*  *d\_work,*  *lwork,*  *d\_rwork,*  *devInfo);*  *cudaStat1 = cudaDeviceSynchronize();*  *assert(CUSOLVER\_STATUS\_SUCCESS == cusolver\_status);*  *assert(cudaSuccess == cudaStat1);* |  |

No Apêndice B.7 consta tabela com os parâmetros de configuração da biblioteca.

## 4.6 – Considerações do Capítulo

Neste capítulo foram comentados os processos adotados e ferramentas para resolução do sistema linear, para programação tradicional e paralelizada. Comentado o uso das bibliotecas CUBLAS, CUSPARSE e CUSOLVER pertencentes a biblioteca CUDA. Estas bibliotecas são utilizadas pelos métodos diretos (LU, Cholesky e QR) e para os métodos iterativos (Gauss Jacobi, Gauss Seidel e SVD). Gauss Jacobi e Gauss Seidel por apresentarem apenas os melhores desempenhos em matrizes de pequenas e médias dimensões, não serão utilizadas para os experimentos. No Apêndice C consta o código fonte escrito em linguagem C++, e as bibliotecas para acessar a arquitetura CUDA.

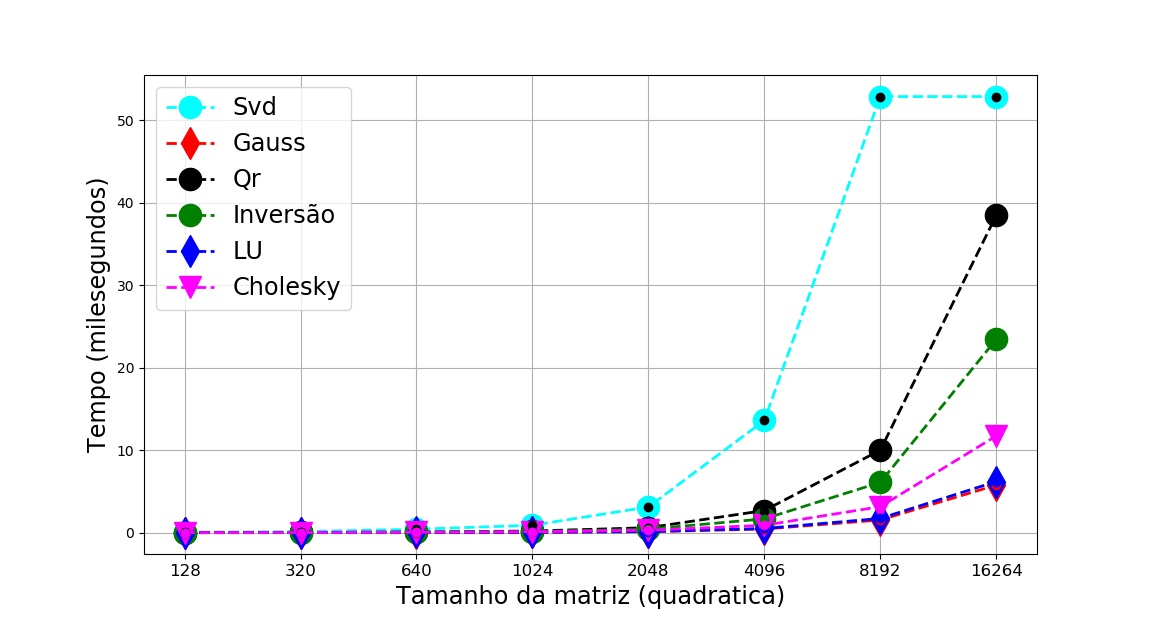
# Capítulo 5 – Resultados

Este capítulo apresenta os resultados obtidos nos experimentos desenvolvidos neste trabalho. Foram utilizados algoritmos para a linguagem C++, inicialmente utilizadas apenas para o processamento em CPU. Posteriormente foram utilizados algoritmos para a GPU, estes códigos foram disponibilizados no github (GITHUB, 2020)*.* Para maior clareza nos gráficos a Eliminação Gaussiana será referenciada apenas como Gauss.

## 5.1 – Avaliar e analisar métodos diretos e iterativos no CPU

O desempenho dos experimentos executados na CPU, são processados 100 vezes para os algoritmos SVD, Gauss, QR, LU e Cholesky. A Inversão da matriz também é utilizada. Todos utilizam o OpenMP.

Figura 20 – Resolução de Sistemas de equações lineares com OpenMP

****

Fonte: Elaborada pelos autores (2020)

Na Figura 20 para a ordem de matriz até 1024, não existe uma diferença expressiva nos algoritmos. Na ordem de matriz de 2048, o algoritmo SVD tem o valor em milissegundos elevado, os demais algoritmos não apresentam diferenças expressivas.

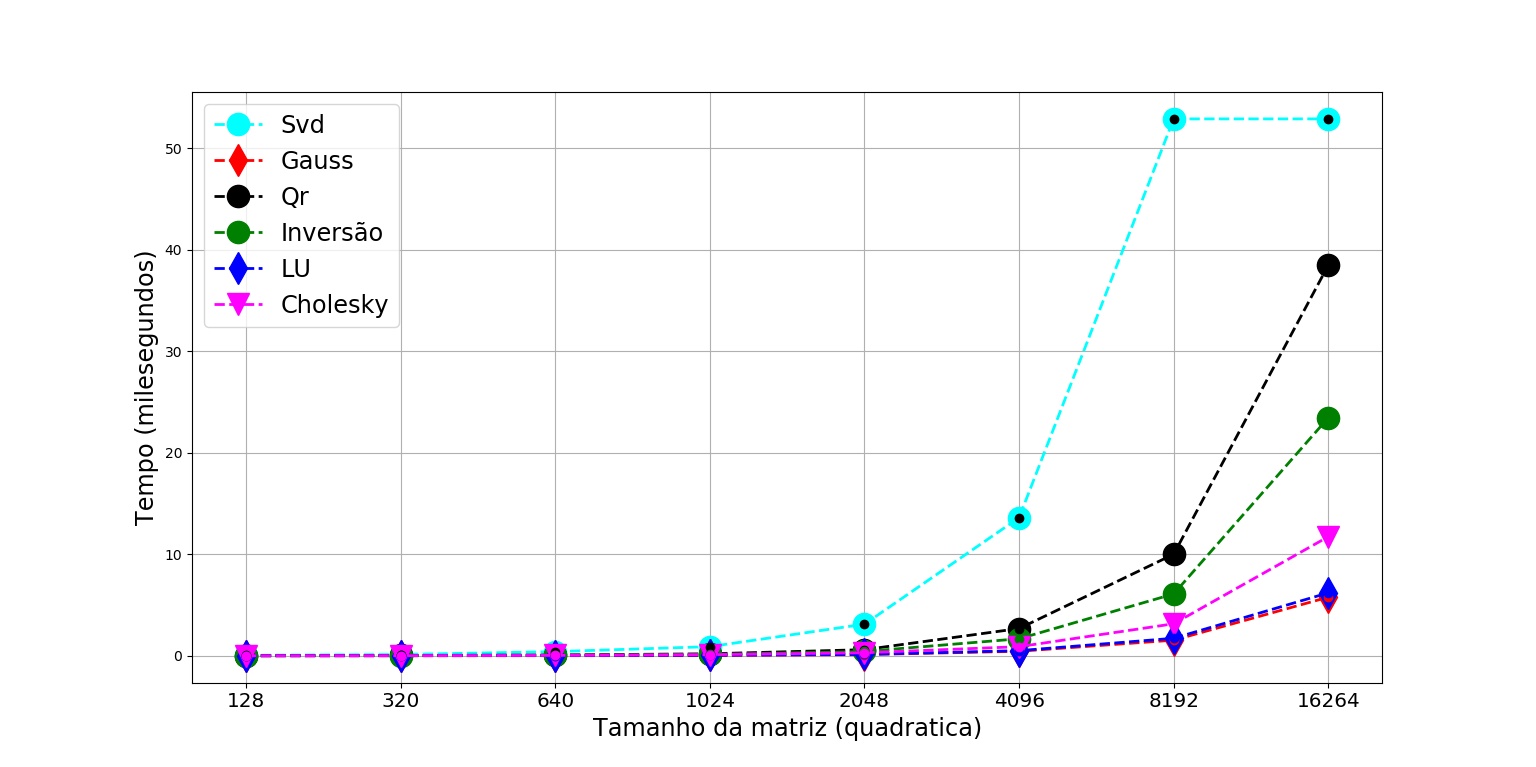
Para a ordem de matriz de 4096, os algoritmos SVD e QR apresentam uma diferença significativa em relação aos demais algoritmos. Na ordem de matriz de 8192, os algoritmos SVD, QR e a inversão de matriz apresentam um crescimento exponencial, com destaque para o SVD.

Ao atingir a ordem de matriz de 16264, para o algoritmo SVD não foi mais possível ser simulado na máquina utilizada, desta forma o valor foi repetido da matriz anterior. Os algoritmos QR, inversão de matriz e Cholesky apresentam um crescimento exponencial. Os algoritmos Gauss e LU apresentam também um crescimento, porém não tão expressivos quanto em comparação com os demais.

## 5.2 – Avaliar e analisar métodos diretos e iterativos na GPU

O desempenho dos experimentos executados na GPU, são processados 100 vezes para os algoritmos SVD, Gauss, QR, LU e Cholesky. A Inversão da matriz também é utilizada. Todos utilizam CUDA.

Figura 21 – Resolução de Sistemas de equações lineares com CUDA

****

Fonte: Elaborada pelos autores (2020)

Na Figura 21 para a ordem de matriz até 2048, não existe uma diferença expressiva nos algoritmos. Exceto o algoritmo SVD o qual tem valor em milissegundos elevado. Na ordem de matriz de 4096, os algoritmos SVD e QR apresentam uma diferença significativa em relação aos demais algoritmos, com destaque para o SVD.

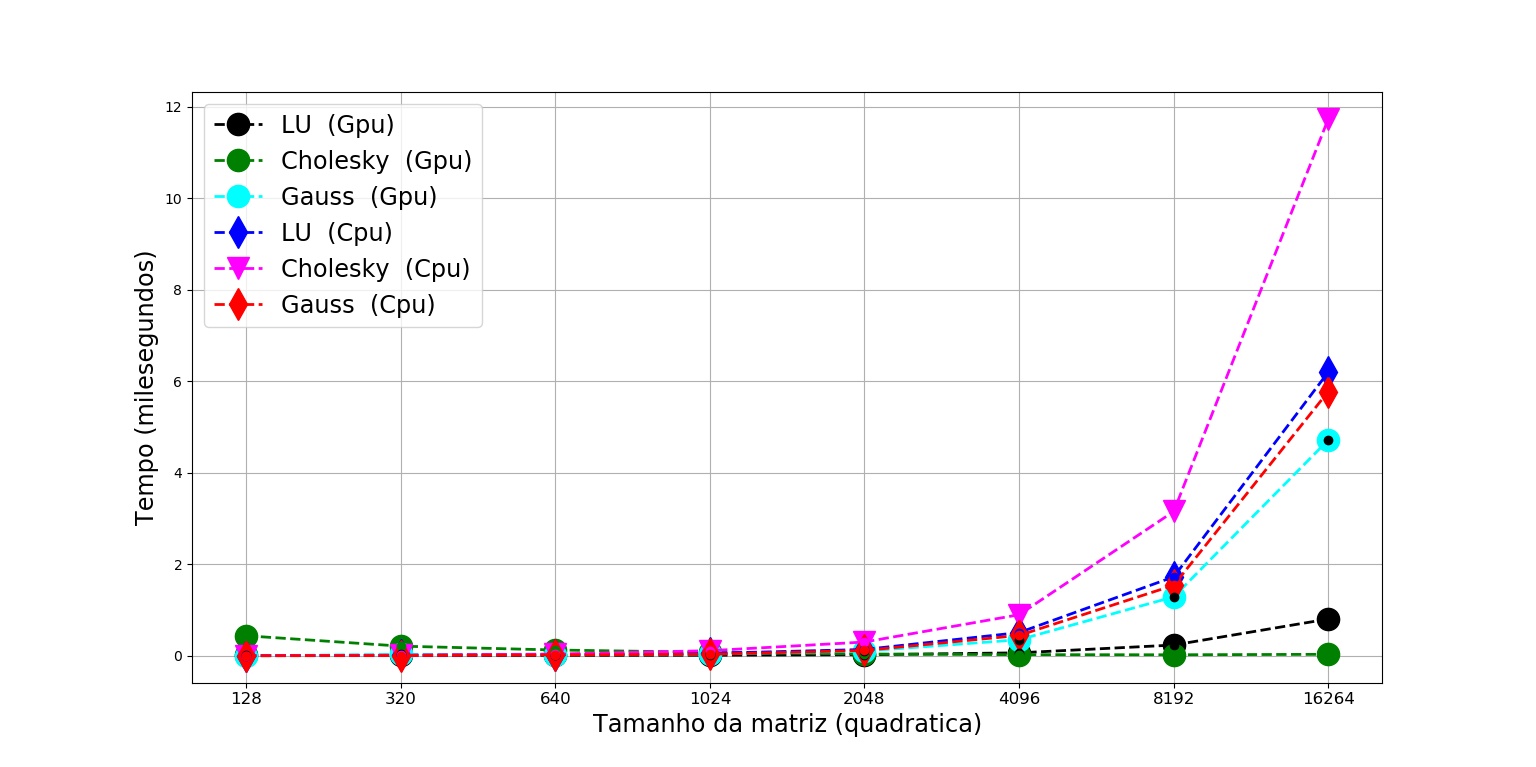
Para a ordem de matriz de 8192, os algoritmos SVD e QR apresentam crescimento exponencial. A inversão da matriz e o algoritmo Cholesky apresenta crescimento substancial em milissegundos. Os algoritmos Gauss e LU apresentam também crescimento, porém não tão expressivos quanto em comparação com os demais.

Ao atingir ordem de matriz de 16264, para o algoritmo SVD, novamente não foi possível ser simulado, o valor foi repetido da matriz anterior. Os algoritmos QR e a inversão de matriz apresentam crescimento exponencial. O algoritmo Cholesky apresenta crescimento substancial em milissegundos. Os algoritmos LU e Gauss não apresentam crescimentos expressivos quando em comparação com os demais.

## 5.3 – Comparar os experimentos executados na CPU x GPU

O desempenho dos experimentos executados na CPU e GPU, são comparados para avaliar os melhores algoritmos utilizados em OpenMP e CUDA.

Figura 22 – Comparativo OpenMP x CUDA



Fonte: Elaborada pelos autores (2020)

Na figura 22 foram observadas as medianas em milissegundos dos algoritmos, e desta forma selecionados os 3 melhores. Pelo processamento em CPU, os algoritmos diretos foram escolhidos. Para o processamento em GPU, os mesmos algoritmos foram selecionados.

Para a ordem de matrizes até 2048, não existe diferença expressiva nos algoritmos. Exceto o algoritmo Cholesky em CPU, o qual tem valor em milissegundos elevado. Na ordem de matriz de 4096, os algoritmos LU, Cholesky e Gauss, ambos em CPU, tem valor em milissegundos elevado.

Na ordem de matriz de 8192, os algoritmos LU, Cholesky e Gauss, ambos em CPU, apresentam crescimento exponencial. Gauss em GPU apresenta crescimento substancial em milissegundos.

Ao atingir ordem de matriz de 16264, os algoritmos LU, Cholesky e Gauss, ambos em CPU, apresentam os maiores crescimentos exponenciais. Gauss em GPU também apresenta crescimento exponencial, porém ainda apresenta melhores resultados que os algoritmos em CPU. O algoritmo LU em GPU apresenta valor em milissegundos mais elevado, quando em comparação com o algoritmo Cholesky em GPU.

**Tabela 4**: Tempo processamento CPU x GPU em milissegundos

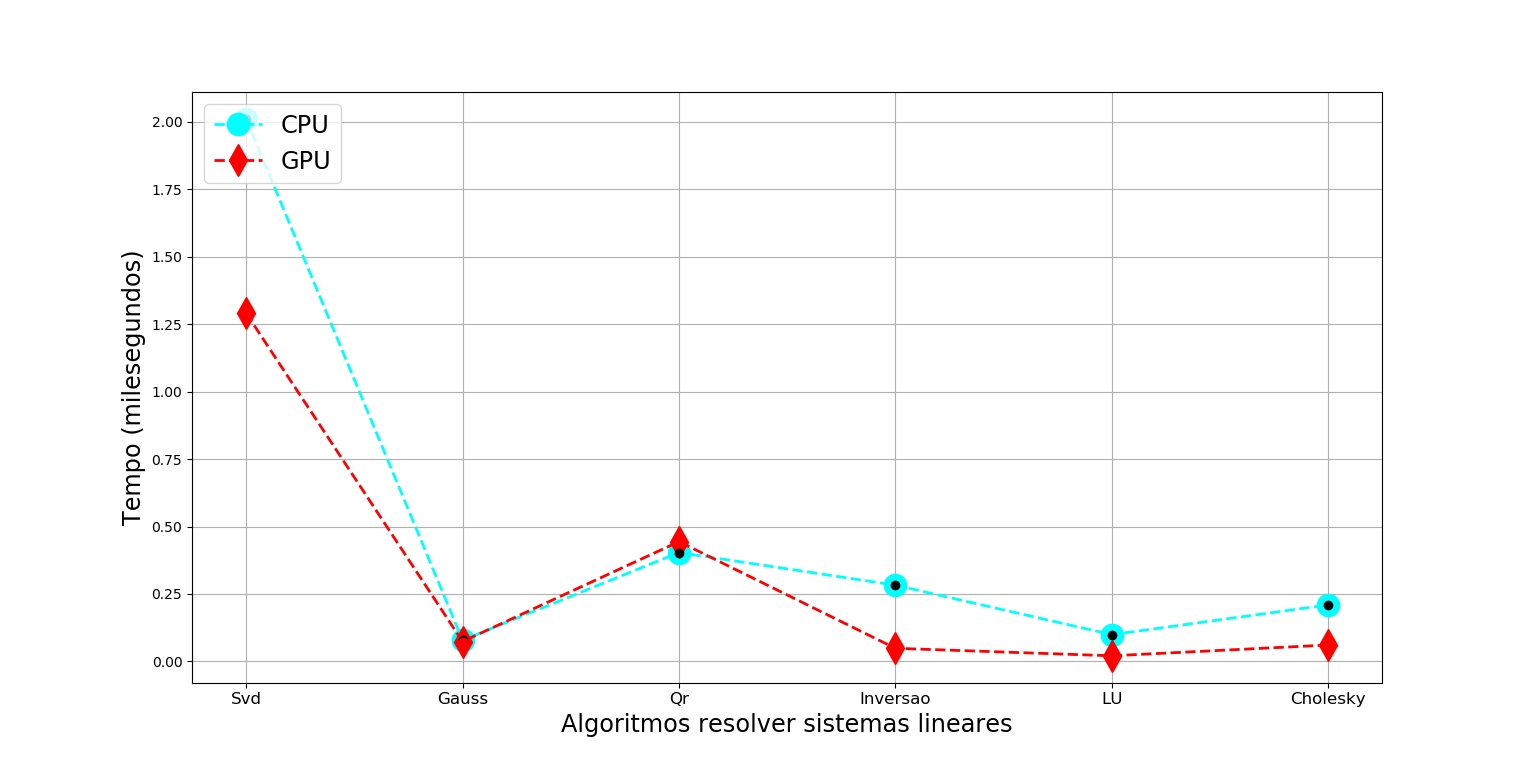
|  |  |  |  |
| --- | --- | --- | --- |
| Algoritmos | Linguagem  Tradicional em CPU | Linguagem com GPU | Diferença |
| SVD | 2,0113 | 1,2911 | 0,7202 |
| Gauss | 0,0787 | 0,0737 | 0,005 |
| QR | 0,4037 | 0,4441 | -0,0404 |
| Inversão Matriz | 0,2826 | 0,0487 | 0,2339 |
| LU | 0,0992 | 0,0209 | 0,0783 |
| Cholesky | 0,2096 | 0,0609 | 0,1487 |
| **Mediana** | **0,5141** | **0,3232** |  |

**Fonte**: Elaborada pelos autores (2020)

Na Tabela 4 são demonstradas as medianas de cada algoritmo, para as matrizes quadráticas utilizadas com seus respectivos tempos. As mesmas matrizes são processadas tanto em programação tradicional usando CPU quanto em programação paralela usando GPU.

O algoritmo SVD apresentou tempo maior para a CPU em comparação com a GPU, nem todas as matrizes puderam serem utilizadas.

Para o algoritmo Gauss os tempos não apresentaram diferenças expressivas em CPU e GPU. O algoritmo QR apresentou melhor desempenho em CPU. A Inversão da matriz apresentou melhores resultados na GPU. Os algoritmos LU e Cholesky apresentaram melhores resultados em GPU. Desta forma, o custo computacional dos algoritmos é importante ser levado em consideração, para a avaliação dos recursos necessários e destinados para o experimento.

Figura 23 – Custo computacional do algoritmo 

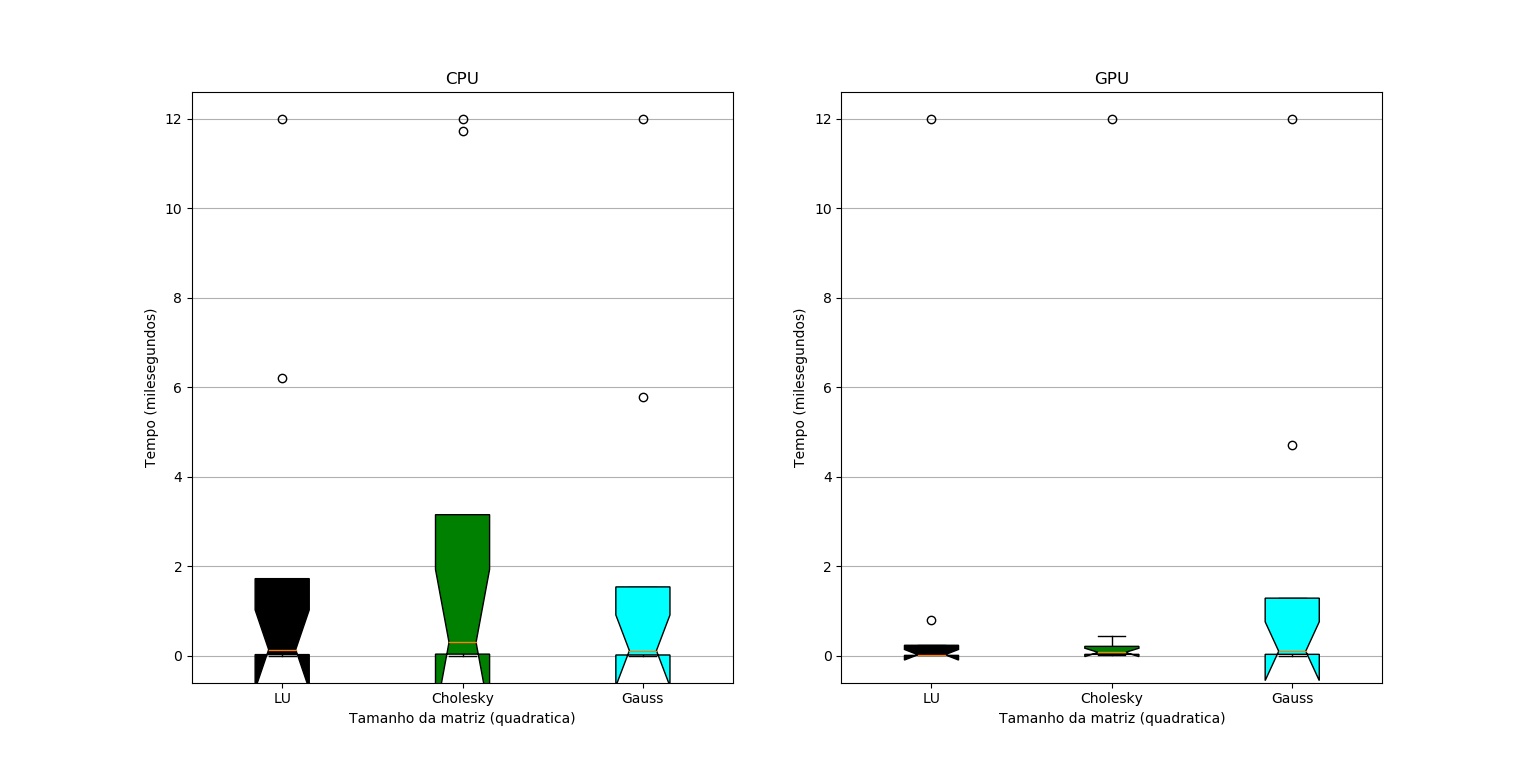
Fonte: Elaborada pelos autores (2020)

Na Figura 23 temos a curva do custo computacional, com o respectivo comportamento e *speedup* de cada uso da linguagem tradicional e paralela. Utilizando as funções CUDA para realizar as operações nas matrizes, o tempo médio do cálculo destas matrizes foi reduzido em 62,86%.

## 5.4 – Comparar a paralelização em diferentes GPUs

Verificar o comportamento dos algoritmos executados na CPU e GPU A, é necessário para possibilitar estimativa do real ganho computacional, quando estes são expostos a diferentes tamanhos de matrizes. Também é necessário checar o ganho computacional, quando estes forem expostos a outros *devices*, com maior poder computacional, aqui representados pela GPU B e GPU C.

Figura 24 – Comparativo algoritmos de CPU e GPU A

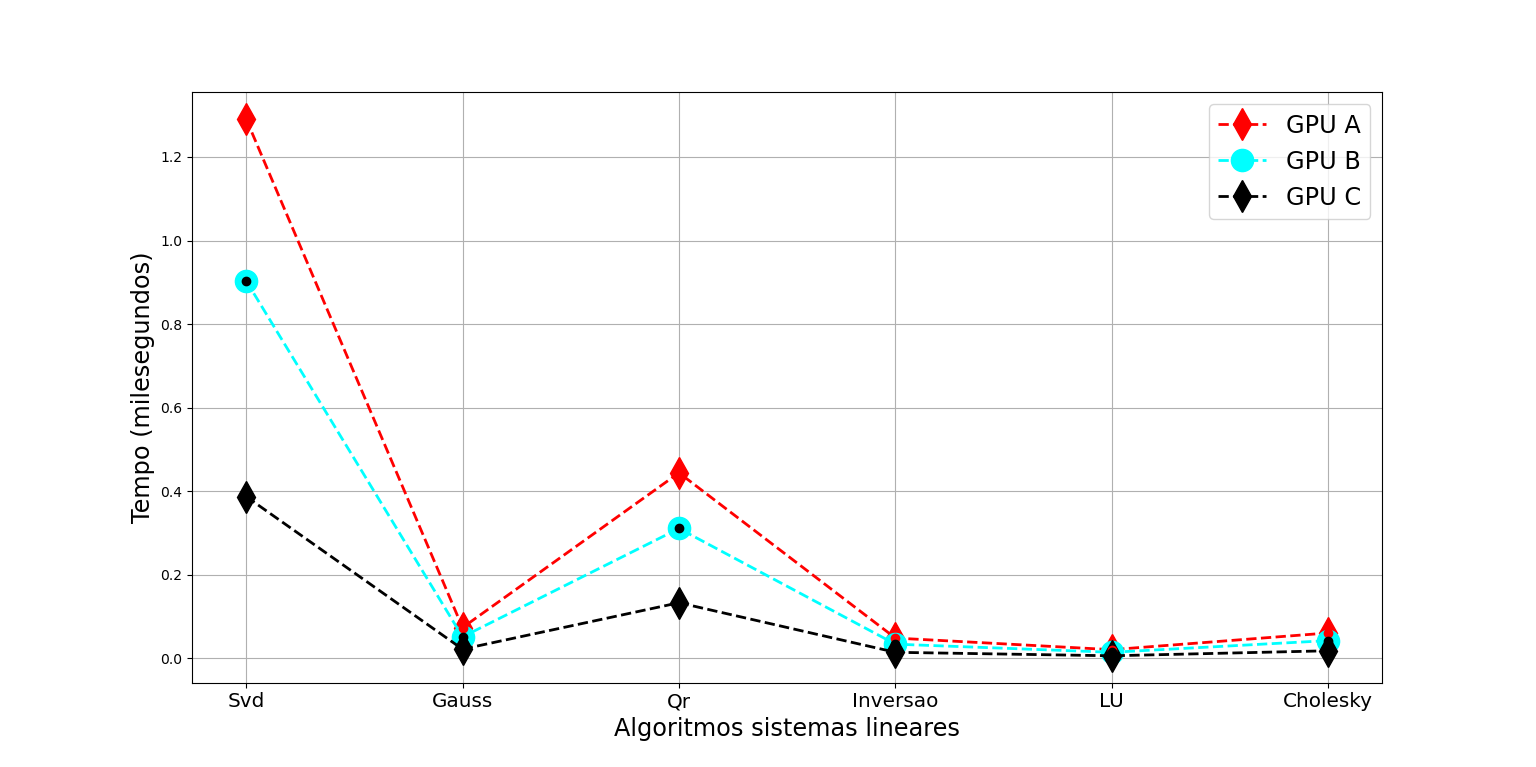


Fonte: Elaborada pelos autores (2020)

Na Figura 24 analisamos o comportamento das medianas dos algoritmos LU, Cholesky e Gauss. Levantou-se os tamanhos de matrizes usados na tabela 4. Observa-se que para a CPU existe uma dispersão maior, conforme os tamanhos das matrizes aumentam, isto em comparação a GPU A.

O algoritmo LU na CPU apresenta uma variação aproximada a 2X, quando comparada a sua execução na GPU. O algoritmo Cholesky na CPU apresenta o maior desbalanceamento ao aumentar o tamanho das matrizes, incremento próximo de 4X em relação a GPU. O algoritmo Gauss em CPU e GPU apresenta uma variação pouco significante para ambos.

Figura 25 – Custo computacional dos algoritmos na GPU A, B e C



Fonte: Elaborada pelos autores (2020)

Na Figura 25 a inversão da matriz e os algoritmos SVD, Gauss, Qr, LU e Cholesky são executados nas GPU A, B e C.

Na GPU B ao comparar com à GPU A, já anteriormente analisada, percebe-se um melhor *speedup*, isto devido ao poder computacional da GPU B (valor computacional de 6.1) ser maior que dá GPU A (valor computacional de 5.0).

O algoritmo SVD apresenta percentual de 30% melhor em comparação com a GPU A. Os demais algoritmos também apresentam melhores resultados computacionais, porém com menores valores de *speedup*.

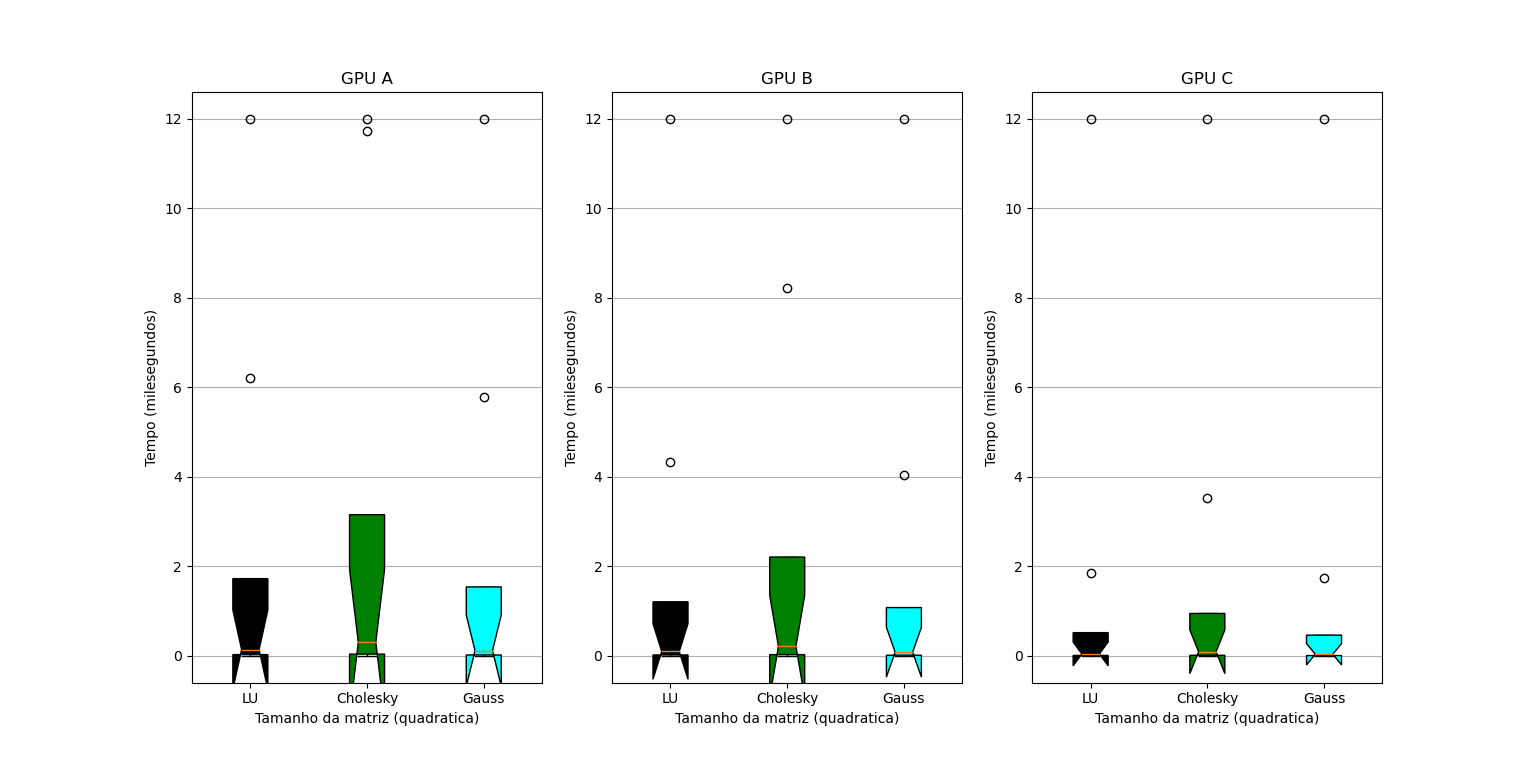
Na GPU C (valor computacional de 7.5) ocorre a mesma tendência, quando comparada com as demais GPUs. Os algoritmos têm *speedup* correlacionado ao valor computacional.

O algoritmo SVD apresenta percentual de 70% melhor em comparação com a GPU A. O algoritmo Gauss teve resultados próximos em ambas as GPUs, com melhores resultados na GPU C.

O algoritmo QR mesmo com resultados insatisfatórios nas GPUs obteve melhor desempenho na GPU C. A inversão da matriz não apresentou resultados relevantes em ambas as GPUs.

O algoritmo LU apresentou uma tendência de melhores resultados computacionais, conforme exposto a maior poder computacional o *speedup* teve aumento. O algoritmo Cholesky também convergiu no mesmo sentido de aumento de *speedup* em relação as GPU A e GPU B.

Figura 26 – Comparativo dos algoritmos na GPU A, B e C



Fonte: Elaborada pelos autores (2020)

Na Figura 26 analisamos o comportamento das medianas dos algoritmos LU, Cholesky e Gauss. Observa-se menor dispersão dos algoritmos na GPU C em relação a GPU A e B. Na GPU B apresenta uma dispersão maior que GPU A e inferior a GPU C.

**Tabela 5**: Tempo processamento diferentes GPUs em milissegundos

|  |  |  |  |
| --- | --- | --- | --- |
| Algoritmos | GPU A | GPU B | GPU C |
| SVD | 1,2911 | 0,9037 | 0,3873 |
| Gauss | 0,0737 | 0,0516 | 0,0221 |
| QR | 0,4441 | 0,3108 | 0,1332 |
| Inversão Matriz | 0,0487 | 0,0341 | 0,0146 |
| LU | 0,0209 | 0,0144 | 0,0062 |
| Cholesky | 0,0609 | 0,0426 | 0,0182 |
| **Mediana** | **0,3232** | **0,2262** | **0,0969** |

**Fonte**: Elaborada pelos autores (2020)

A Tabela 5 demonstra as medianas dos algoritmos exposto as GPU A, B e C. São observados os tempos para todos os tamanhos de matrizes utilizados anteriormente.

## 5.5 – Considerações do Capítulo

Neste capítulo foram comparados os processos pelos algoritmos SVD, Gauss, QR, LU, Cholesky e por Inversão da Matriz. Os métodos diretos Gauss, LU e Cholesky apresentaram melhores resultados comparados aos métodos iterativos. Os resultados mostraram que mesmo realizando uma pequena parte do processamento na GPU, e com a combinação da linguagem interpretada no processamento na GPU, oferece alternativa bastante conveniente para o desenvolvimento de algoritmos numéricos.

# Capítulo 6 – Conclusões e trabalhos futuros

Nesse trabalho, utilizamos à análise nodal modificada em outra abordagem computacional, uma estratégia de processamento paralelo em GPU. Representamos os algoritmos solucionadores comparados em CPU e GPUs. Utilizamos as bibliotecas CUBLAS, CUSPARSE e CUSOLVER pertencentes a arquitetura CUDA para acessar GPUs. O benefício da utilização de GPUs e que estas podem incorporar mais gigabytes DRAM, e os *kernels* CUDA podem operar sobre dados muito grandes, sejam eles hospedados inteiramente na DRAM da GPU ou acessando a memória mapeada do host.

A memória física do sistema da CPU e da GPU, podem ser mapeadas dentro de um único espaço de endereços virtuais compartilhados. Espaço de endereços globais compartilhados, permitiram que todas as variáveis da aplicação tivessem endereços exclusivos, o acesso diretamente a memorias físicas possibilitaram aceleração das aplicações pela GPU.

O espaço do endereço global compartilhado permitiu transferência de dados direta entre GPUs, permitindo transferir dados diretamente para dentro e fora da memória física da GPU, reduziu significativamente o custo de cópia, e a apresentou melhora no desempenho das aplicações que processaram grandes conjuntos de dados, possibilitando eficiência da largura de banda da memória.

Com maior eficiência da largura de banda da memória e com operações atômicas mais rápidas, foi possível reduzir a necessidade de envolver a CPU host. Algoritmos realizaram operações coletivas, onde múltiplos blocos de *threads* CUDA, reduziram a transferência de dados entre CPU e a GPU. Isto contribuiu para que mais algoritmos para matrizes perfeitamente ladrilhados fossem utilizados diretamente na GPU, usufruindo da técnica de processamento paralelo ajustadas à GPU, para uma computação do tipo dividir para conquistar.

Com a divisão em múltiplos *kernels* da mesma aplicação, foram executados simultaneamente para utilizar a GPU por completo, onde o cluster em paralelo segmentou o trabalho em partições, e em tamanhos de *kernels* muito menores apresentando melhores resultados para o trabalho.

Os resultados obtidos demonstram que o processamento paralelo tem menor impacto computacional quando comparados os algoritmos dos experimentos 1 e 2. Estes se referem a computação em CPU e GPU respectivamente, para a resolução de sistemas lineares com números reais e complexos. No experimento 3 os melhores resultados destes experimentos foram analisados, observando ganho computacional para os experimentos em GPU. Os métodos de análise nodal modificada permitem a utilização de programação em *threads,* e a independência dos dados possibilitam que a resolução seja executada em tarefas paralelizadas. Os desempenhos nas GPUs apresentam 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 à 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. Conclui-se que a abordagem paralela reduz o tempo computacional para circuitos típicos em comparação com processamento não paralelizado. Contribuindo para minimizar o impacto no desempenho do desenvolvimento de um Gêmeo Digital.

Sugere-se como trabalho futuro a análise do desempenho em outras GPUs de maior porte computacional, utilizando outras técnicas de programação, verificando a possibilidade de melhorar o desempenho, visto que existe franca evolução destes *devices*. Analisar o impacto do *hardware* na resolução de sistemas lineares, considerando que as técnicas de programação impactam de forma considerável, tendo em vista os resultados apresentados.

# Referências Bibliográficas

ALBA-FLORES, R. **Circuit Analysis Lab**, Amazon Digital Services LLC, 2013.

ABUSALAH, A. et al. Accelerated Sparse Matrix-Based Computation of Electromagnetic Transients. **IEEE Open Access Journal of Power and Energy**, v. 7, p. 13-21, 2019.

DOI: 10.1109/OAJPE.2019.2952776. Acesso em: 22 de janeiro de 2020.

ALEXANDER, C; SADIKU, M.  **Fundamentals of electric circuits**. 5º Ed., McGraw-Hill, 2017.

AMD. AMD Accelerated Parallel Processing: **OpenCL Zone – Accelerated Parallel Processing**. AMD Corporation, 2016. Disponível em: <http://developer.amd.com/wordpress/media/2013/07/AMD_Accelerated_Parallel_Processing_OpenCL_Programming_Guide-rev-2.7.pdf>. Acesso em: 22 de janeiro de 2020.

AMDAHL, Gene M. Validity of the single processor approach to achieving large scale computing capabilities. In: **Proceedings of the April 18-20, 1967, spring joint computer conference**. 1967. p. 483-485. Disponível em: https://doi.org/10.1145/1465482.1465560. Acesso em: 22 de janeiro de 2020.

AMDAHL, Gene Myron. **The logical design of an intermediate speed digital computer**, unpublished PhD dissertation, Univ. of Wisconsin, Madison, Wisconsin, Estados Unidos, 1952.

Disponível em: http://pages.cs.wisc.edu/~bezenek/Stuff/amdahl\_thesis.pdf. Acesso em: 22 de janeiro de 2020.

ANNAVARAM, Murali; GROCHOWSKI, Ed; SHEN, John. Mitigating Amdahl's law through EPI throttling. In: **32nd International Symposium on Computer Architecture (ISCA'05)**. IEEE, 2005. p. 298-309. DOI: [10.1109/ISCA.2005.36](https://doi.org/10.1109/ISCA.2005.36). Acesso em: 22 de janeiro de 2020.

AUGUST, David I. et al. Achieving structural and composable modeling of complex systems. **International Journal of Parallel Programming**, v. 33, n. 2-3, p. 81-101, 2005.

Disponível em: <https://doi.org/10.1007/s10766-005-3569-3>. Acesso em: 22 de janeiro de 2020.

ARDEN, Wolfgang et al. More-than-Moore white paper. **Version**, v. 2, p. 14, 2010.

ASANACHE, Răzvan et al. On Circuit Analysis and Simulation of Networks with Nullors. In: **2019 8th International Conference on Modern Power Systems (MPS)**. IEEE, 2019. p. 1-7.

DOI: [10.1109/MPS.2019.8759698](https://doi.org/10.1109/MPS.2019.8759698). Acesso em: 22 de janeiro de 2020.

AZ, Ilgaz; SAHIN, Suhap; CAVUSLU, Mehmet Ali. Implementation of fast fourier and inverse fast fourier transforms in FPGA. In: **2007 IEEE 15th Signal Processing and Communications Applications**. IEEE, 2007. p. 1-4.

BAKHODA, Ali et al. Analyzing CUDA workloads using a detailed GPU simulator. In: **2009 IEEE International Symposium on Performance Analysis of Systems and Software**. IEEE, 2009. p. 163-174. Disponível em: <https://doi.org/10.1109/ispass.2009.4919648>. Acesso em: 22 de janeiro de 2020.

BOYLESTAD, Robert L. **Introductory circuit analysis**, 11º. Pearson Education, 2015.

CALAHAN, Donald A. **Computer-Aided Network Design: Revised Edition**. MICHIGAN UNIV ANN ARBOR DEPT OF ELECTRICAL ENGINEERING, 1972.

CHAPMAN, Barbara; JOST, Gabriele; VAN DER PAS, Ruud. **Using OpenMP: portable shared memory parallel programming**. MIT press, 2008.Scientific and Engineering Computation, ISBN-13: 978-0-262-53302-7.

CHRZESZCZYK, A.; CHRZESZCZYK, J. Matrix computations on the GPU CUBLAS and MAGMA by example. **Retrieved**, v. 16, p. 2017, 2015, 2013. Acesso em: 22 de janeiro de 2020.

CHEW, Alvin Wei Ze; LAW, Adrian Wing-Keung; VU, Tung Thanh. Optimizing Speedup Performance of Computational Hydrodynamic Simulations with UPC Programming Model. **Journal of Computing in Civil Engineering**, v. 34, n. 2, p. 06020001, 2020.

DOI: 10.1061/(ASCE) CP.1943-5487.0000876. Acesso em: 22 de janeiro de 2020.

CHHABRIA, Nikhil et al. RLC circuit simulation and Monte Carlo Analysis in MATLAB. In: **2016 International Conference on Communication and Electronics Systems (ICCES)**. IEEE, 2016. p. 1-6.

CORRÊA, Gustavo Rosa; SULZBACH, Maurício. PROGRAMAÇÃO PARALELA EM CPU E GPU: UMA AVALIAÇÃO DO DESEMPENHO DAS APIS OPENMP, CUDA, OPENCL E OPENACC. **RECeT-Revista de Engenharia, Computação e Tecnologia**, v. 1, n. 1, 2017.

Disponível em: http://revistas.fw.uri.br/index.php/recet/article/view/2261/pdf\_1. Acesso em: 22 de janeiro de 2020.

CUDA, Nvidia. Compute unified device architecture programming guide. 2007, version 2.3. [Online].

Disponível em: https://docs.nvidia.com/cuda/pdf/CUDA\_C\_Programming\_Guide.pdf, 2009.

Acesso em: 22 de janeiro de 2020.

CUDA, Nvidia. GPU recomendada para desenvolvedores. 2009. [Online].

Disponível em: <https://developer.nvidia.com/cuda-gpus#compute>, 2009.

Acesso em: 22 de janeiro de 2020.

DING, Chen. A Symmetric Successive Overrelaxation (SSOR) based Gauss-Seidel Massive MIMO Detection Algorithm. In: **Journal of Physics: Conference Series**. IOP Publishing, 2020. p. 012005. Acesso em: 22 de janeiro de 2020.

DUFF, Iain; HOGG, Jonathan; LOPEZ, Florent. A New Sparse LDL^T Solver Using A Posteriori Threshold Pivoting. **SIAM Journal on Scientific Computing**, v. 42, n. 2, p. C23-C42, 2020. Acesso em: 22 de janeiro de 2020.

EL SADDIK, Abdulmotaleb. Digital twins: The convergence of multimedia technologies. IEEE MultiMedia, v. 25, n. 2, p. 87-92, 2018.

DOI: 10.1109/MMUL.2018.023121167. Acesso em: 22 de janeiro de 2020.

FEITELSON, Dror G. The Effect of Metrics and Workloads on the Evaluation of Computer Systems. Technical Report 2001, The Hebrew University of Jerusalem, Oct 2001.

Disponível em: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.18.7252&rep=rep1&type=pdf. Acesso em: 22 de janeiro de 2020.

FREY, Dietmar et al. A Precision Medicine Framework for Personalized Simulation of Hemodynamics in Cerebrovascular Disease. **medRxiv**, 2020. DOI: 10.1101/2020.01.28.20019190. Acesso em: 22 de janeiro de 2020.

GAMMA, Erich et al. Design Patterns: Elements of Reusable Object-Oriented Software Addison-Wesley. **Reading, MA**, v. 2, p. 369-378, 1995.

Disponível em: http://www.uml.org.cn/c++/pdf/designpatterns.pdf. Acesso em: 22 de janeiro de 2020.

GASSIAT, Paul et al. Speed of propagation for Hamilton–Jacobi equations with multiplicative rough time dependence and convex Hamiltonians. **Probability Theory and Related Fields**, v. 176, n. 1, p. 421-448, 2020. Acesso em: 28 de janeiro de 2020.

GEAR, C. W. The automatic integration of stiff ordinary differential equations. **Proc. 68**, p. 187-193, 1969.

Disponível em: <https://ntrs.nasa.gov/archive/nasa/casi.ntrs.nasa.gov/19680023773.pdf#page=11>. Acesso em: 28 de janeiro de 2020.

GUSTAFSON, John L. Reevaluating Amdahl's law. **Communications of the ACM**, v. 31, n. 5, p. 532-533, 1988. Disponível em: https://www.researchgate.net/profile/Yuan\_Shi12/publication/228367369\_Reevaluating\_Amdahl's\_law\_and\_Gustafson's\_law/links/562f9dd408ae8e1256876a0a/Reevaluating-Amdahls-law-and-Gustafsons-law.pdf. Acesso em: 22 de janeiro de 2020.

GWENNAP, Linley. Adapteva: More flops, less watts. **Microprocessor Report**, v. 6, n. 13, p. 11-02, 2011. Disponível em: https://www.adapteva.com/wp-content/uploads/2011/06/adapteva\_mpr.pdf. Acesso em: 22 de janeiro de 2020.

HACHTEL, Gary; BRAYTON, R.; GUSTAVSON, F. The sparse tableau approach to network analysis and design. **IEEE Transactions on circuit theory**, v. 18, n. 1, p. 101-113, 1971.

Disponível em: https://doi.org/10.1109/TCT.1971.1083223. Acesso em: 22 de janeiro de 2020.

HASHEMIAN, Reza. UaL Decomposition, an Alternative to the LU Factorization of MNA Matrices. **IEEE Transactions on Circuits and Systems II: Express Briefs**, 2019.

DOI: 10.1109/TCSII.2019.2924898. Acesso em: 22 de janeiro de 2020.

HARTMANN, Dirk; VAN DER AUWERAER, Herman. Digital Twins. arXiv preprint arXiv:2001.09747, 2020. Acesso em: 22 de janeiro de 2020.

HAYT, William Hart; KEMMERLY, Jack Ellsworth; DURBIN, Steven M. **Engineering circuit analysis**. New York: McGraw-Hill, 2018.

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

HO, Chung-Wen; RUEHLI, A.; BRENNAN, Pierce. The modified nodal approach to network analysis. **IEEE Transactions on circuits and systems**, v. 22, n. 6, p. 504-509, 1975.

Disponível em: https://doi.org/10.1109/TCS.1975.1084079. Acesso em: 22 de janeiro de 2020.

IRWIN, J. David; NELMS, Robert M. **Basic engineering circuit analysis**, 10ª Ed., Wiley, 2015.

JANUSZEWSKI, Michal; KOSTUR, Marcin. Accelerating numerical solution of stochastic differential equations with CUDA. **Computer Physics Communications**, v. 181, n. 1, p. 183-188, 2010.

Disponível em: <https://doi.org/10.1016/j.cpc.2009.09.009>. Acesso em: 22 de janeiro de 2020.

KAGERMANN, Henning et al. **Recommendations for implementing the strategic initiative INDUSTRIE 4.0: Securing the future of German manufacturing industry; final report of the Industrie 4.0 Working Group**. Forschungsunion, 2013. Acesso em: 22 de janeiro de 2020.

KANOUSSIS, D. **Analysis of Electric Circuits**, Amazon Digital Services LLC, 2017.

KOSINSKI, A. A. Cramer's rule is due to Cramer. **Mathematics Magazine**, v. 74, n. 4, p. 310-312, 2001.

KRITZINGER, Werner et al. Digital Twin in manufacturing: A categorical literature review and classification. **IFAC-PapersOnLine**, v. 51, n. 11, p. 1016-1022, 2018.

Doi: [10.1016/j.ifacol.2018.08.474](https://doi.org/10.1016/j.ifacol.2018.08.474). Acesso em: 22 de janeiro de 2020.

LEE, Edward A. Cyber physical systems: Design challenges. In: **2008 11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC)**. IEEE, 2008. p. 363-369. DOI: 10.1109/ISORC.2008.25. Acesso em: 22 de janeiro de 2020.

LEW-YEE, Juan Felipe Huan et al. Asymmetric Density Fitting with Modified Cholesky Decomposition Applied to Second-Order Electron Propagator. **Journal of Chemical Theory and Computation**, v. 16, n. 3, p. 1597-1605, 2020.

LINDHOLM, Erik et al. NVIDIA Tesla: A unified graphics and computing architecture. **IEEE micro**, v. 28, n. 2, p. 39-55, 2008. DOI: [10.1109/mm.2008.31](https://doi.org/10.1109/mm.2008.31). Acesso em: 22 de janeiro de 2020.

LI, Xiuhong et al. A coordinated tiling and batching framework for efficient GEMM on GPUs. In: **Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming**. 2019. p. 229-241. Acesso em: 22 de janeiro de 2020.

LIN, Ding-Bing; WANG, Tse-Hsuan; CHIANG, Jui-Wen. Suppression of Wideband Simultaneous Switching Noise Through Application of a Partial Electromagnetic Band-Gap Structure in Multilayer Printed Circuit Boards. **IEEE Transactions on Electromagnetic Compatibility**, 2019. Acesso em: 22 de janeiro de 2020.

LYCHE, Tom. Gaussian Elimination and LU Factorizations. In: **Numerical Linear Algebra and Matrix Factorizations**. Springer, Cham, 2020. p. 57-81. Acesso em: 22 de janeiro de 2020.

MAGDOWSKI, Mathias; VICK, Ralf. Transient Simulation of the Plane Wave Coupling to Non-Linearly Loaded Transmission Line Networks. In: **2019 Joint International Symposium on Electromagnetic Compatibility, Sapporo and Asia-Pacific International Symposium on Electromagnetic Compatibility (EMC Sapporo/APEMC)**. IEEE, 2019. p. 383-386. DOI: [10.23919/EMCTokyo.2019.8893805](https://doi.org/10.23919/EMCTokyo.2019.8893805). Acesso em: 22 de janeiro de 2020.

MATLOFF, N. Programming on Parallel Machines University of California: Davis, 2012. **URL: http://heather. cs. ucdavis. edu/~ matloff/158/PLN/ParProcBook. pdf**. Acesso em: 22 de janeiro de 2020.

MIRZAHOSSEINI, Ramin; IRAVANI, Reza. Small Time-Step FPGA-Based Real-Time Simulation of Power Systems Including Multiple Converters. **IEEE Transactions on Power Delivery**, v. 34, n. 6, p. 2089-2099, 2019. DOI: 10.1109/TPWRD.2019.2933610. Acesso em: 22 de janeiro de 2020.

MIVULE, Kato et al. A review of cuda, mapreduce, and pthreads parallel computing models. **arXiv preprint arXiv:1410.4453**, 2014. Disponível em: <https://arxiv.org/ftp/arxiv/papers/1410/1410.4453.pdf>. Acesso em: 22 de janeiro de 2020.

MOLNAR JR, Ferenc et al. Air pollution modelling using a Graphics Processing Unit with CUDA. **Computer Physics Communications**, v. 181, n. 1, p. 105-112, 2010.

DOI: 10.1016/j.cpc.2009.09.008. Acesso em: 22 de janeiro de 2020.

NAEINI, Amin Alizadeh et al. Application of PCA Analysis and QR Decomposition to Address RFM's Ill-Posedness. **Photogrammetric Engineering & Remote Sensing**, v. 86, n. 1, p. 17-21, 2020. Acesso em: 22 de janeiro de 2020.

NAGEL, Laurence; ROHRER, Ronald. Computer analysis of nonlinear circuits, excluding radiation (CANCER). **IEEE Journal of Solid-State Circuits**, v. 6, n. 4, p. 166-182, 1971.

NAHVI, M; EDMINISTER, J. **Schaum's Outline of Electric Circuits**, 7ª Ed., McGraw-Hill, 2017.

NDUKA, O. S. et al. A Robust Augmented Nodal Analysis Approach to Distribution Network Solution. **IEEE Transactions on Smart Grid**, 2019.

DOI: 10.1109/TSG.2019.2948282. Acesso em: 22 de janeiro de 2020.

NVIDIA, CUDA C Programming Guide. **NVIDIA Corporation**, ed. 5.0. [S.l.]: NVIDIA Corporation, 2013. Disponível em: <https://docs.nvidia.com/cuda/archive/9.1/pdf/CUDA_C_Programming_Guide.pdf>.

Acesso em: 22 de janeiro de 2020.

NVIDIA Corporation: cuBLAS Library User’s Guide. Disponível em: <https://docs.nvidia.com/cuda/pdf/CUBLAS_Library.pdf>, 2019. Acesso em: 22 de janeiro de 2020.

NVIDIA Corporation: cuFFT Library User’s Guide. Disponível em:

<https://docs.nvidia.com/cuda/pdf/CUFFT_Library.pdf>, 2019. Acesso em: 22 de janeiro de 2020.

NVIDIA Corporation: CUDA Toolkit 4.2 cuSPARCE Library. Disponível em: <https://developer.download.nvidia.com/compute/DevZone/docs/html/CUDALibraries/doc/CUSPARSE_Library.pdf>, 2012. Acesso em: 22 de janeiro de 2020.

NVIDIA Corporation: cuSOLVER Library User’s Guide. Disponível em: <https://docs.nvidia.com/cuda/pdf/CUSOLVER_Library.pdf>, 2019. Acesso em: 22 de janeiro de 2020.

NVIDIA. CUDA Zone. **NVIDIA Corporation,** 2016. Disponível em:

<https://developer.nvidia.com/cuda-zone>. Acesso em: 22 de janeiro de 2020.

NVIDIA, “O QUE É COMPUTAÇÃO ACELERADA POR DE PLACAS DE VÍDEO?”

Disponível em: <http://www.nvidia.com.br/object/what-is-gpu-computing-br.html>.

Acesso em: 22 de janeiro de 2020.

O'MALLEY, J. **Basic Circuit Analysis**, 2º Ed., McGraw-Hill, 2011.

OPENMP. **Application Program Interface**. <https://www.openmp.org/>. Disponível em: <http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf>. Acesso em: 22 de janeiro de 2020.

[OPENACC. **Application Programming Interface standard** version 2.5"](https://www.openacc.org/sites/default/files/inline-files/OpenACC_2pt5_0.pdf) (PDF). OpenACC.org, 2015. Disponível em: <https://www.openacc.org/sites/default/files/inline-files/OpenACC_2pt5_0.pdf>. Acesso em: 22 de janeiro de 2020.

PACHECO, André GC. Multiplicação de matrizes: uma comparação entre as abordagens sequencial (CPU) e paralela (GPU). **CoRR**, 2019.

Disponível em: https://arxiv.org/pdf/1905.03641.pdf. Acesso em: 22 de janeiro de 2020.

REIS, Timo; GLAZOV, Fedor. Systems theoretic properties of linear RLC circuits.

PATTERSON, David A.; HENNESSY, John L. **Computer Organization and Design ARM Edition: The Hardware Software Interface**. Morgan kaufmann, 2016.

PENNEC, Xavier. Manifold-valued image processing with SPD matrices. In: **Riemannian Geometric Statistics in Medical Image Analysis**. Academic Press, 2020. p. 75-134.

RAHMANI, Ulfa; PRIBADI, Diantiny Mariam; PURWANI, Sri. Parameter Estimation Using LU Decomposition in the Logistic Regression Model for Credit Scoring Analysis. **World Scientific News**, v. 140, p. 1-11, 2020.

ROBBINS, Allan H.; MILLER, Wilhelm C. **Circuit analysis: Theory and practice**. Cengage Learning, 2012.

RÜDE, Ulrich et al. Research and education in computational science and engineering. **Siam Review**, v. 60, n. 3, p. 707-754, 2018.

RUETSCH, Greg; MICIKEVICIUS, Paulius. Optimizing matrix transpose in CUDA. **Nvidia CUDA SDK Application Note**, v. 18, 2009.

Disponível em: https://www.cs.colostate.edu/~cs675/MatrixTranspose.pdf. Acesso em: 22 de janeiro de 2020.

SALAM, Md Abdus; RAHMAN, Quazi Mehbubar. **Fundamentals of electrical circuit analysis**. New York: Springer, 2018.

SANDERS, Jason; KANDROT, Edward. **CUDA by example: an introduction to general-purpose GPU programming, portable documents**. Addison-Wesley Professional, 2010. Disponível em: https://developer.download.nvidia.com/books/cuda-by-example/cuda-by-example-sample.pdf. Acesso em: 22 de janeiro de 2020.

SCHALLER, Robert R. Moore's law: past, present and future. IEEE spectrum, v. 34, n. 6, p. 52-59, 1997.

SENA, M; COSTA, J. **Tutorial OpenMP C/C++. Programa Campus Ambassador HPC**, SUN Microsystems – Maceió, 2008. Disponível em: http://www.inf.ufrgs.br/~afarah/files/openmp.pdf.

Acesso em: 22 de janeiro de 2020.

SHEPOVALOV, Maxim; AKELLA, Venkatesh. FPGA and GPU-based acceleration of ML workloads on Amazon cloud-A case study using gradient boosted decision tree library. **Integration**, v. 70, p. 1-9, 2020.

DOI: 10.1016/j.vlsi.2019.09.007. Acesso em: 22 de janeiro de 2020.

SHIRI, Imen; LEFTERIU, Sanda; LABARRE, Cécile. Buck Converter Modeling in High Frequency using Several Transfer Function-based Approaches. 2019. DOI: 10.5220/0007927407220729.

Acesso em: 22 de janeiro de 2020.

SIMÕES, M. Godoy; FARRET, Felix A. (Ed.). **Modeling Power electronics and interfacing energy conversion systems**. John Wiley & Sons, 2016.

SMITH, D. Circuit Analysis for Complete Idiots. Amazon Digital Services LLC, 2019. Disponivel em: https://www.amazon.com.br/Circuit-Analysis-Complete-Electrical-Engineering-ebook/dp/B07RXD35JS. Acesso em: 22 de janeiro de 2020.

SOUSA, Gabriel; MACHADO, Bierley Souza; BRODBECK, Leonardo; GALVÃO, Arlindo; CARVALHO, Rafael Viana; COELHO, Clarimar. **Identificação de erros na lista de conectividade de circuito elétrico empregando analisador léxico e sintático**. V Congresso de Ciência e Tecnologia da PUC Goiás, 2019. Disponível em: <http://sites.pucgoias.edu.br/eventos/5cct/>. Acesso em: 22 de janeiro de 2020.

SUTTER, Herb. The free lunch is over: A fundamental turn toward concurrency in software. **Dr. Dobb’s journal**, v. 30, n. 3, p. 202-210, 2005.

Disponível em: http://www.gotw.ca/publications/concurrency-ddj.htm. Acesso em: 23 de janeiro de 2020.

THOMAS, Roland E.; ROSA, Albert J.; TOUSSAINT, Gregory J. **The analysis and design of linear circuits**. John Wiley & Sons, 2011.

VLACH, Jiri. **Linear circuit theory: matrices in computer applications**. Apple Academic Press, 2016. Disponível em: <https://doi.org/10.1201/b16590>. Acesso em: 22 de janeiro de 2020.

WANG, Tse-Hsuan; LIN, Ding-Bing. Using modified nodal analysis in cavity-mode resonances printed circuit board power bus structures with the segmentation method. **Journal of the Chinese Institute of Engineers**, v. 42, n. 6, p. 525-533, 2019. Doi:10.1080/02533839.2019.1611480. Acesso em: 22 de janeiro de 2020.

WIENKE, Sandra et al. OpenACC — first experiences with real-world applications. In: **European Conference on Parallel Processing**. Springer, Berlin, Heidelberg, 2012. p. 859-870.

Disponível em: https://link.springer.com/chapter/10.1007/978-3-642-32820-6\_85. Acesso em: 22 de janeiro de 2020.

YANG, Z. et al. An efficient integration technique for the voxel‐based finite cell method. International Journal for Numerical Methods in Engineering, v. 91, n. 5, p. 457-471, 2012.

ZHENG, Pai et al. Smart manufacturing systems for Industry 4.0: Conceptual framework, scenarios, and future perspectives. **Frontiers of Mechanical Engineering**, v. 13, n. 2, p. 137-150, 2018. Acesso em: 22 de janeiro de 2020.

ZILL, Dennis; WRIGHT, Warren S.; CULLEN, Michael R. Advanced engineering mathematics, 6ª Ed. Jones & Bartlett Learning, 2016.

# Apêndice A - Solucionadores de equações lineares

A.1 – Eliminação Gaussiana

Dado um sistema linear a ser resolvido pelo método de eliminação gaussiana como:

(12)

A matriz estendida da Equação (12), é escrita como:

(13)

No primeiro passo da Equação (13), subtraímos da segunda linha o quádruplo da primeira e subtraímos da terceira linha o dobro da primeira linha:

(14)

No segundo passo da Equação (14), permutamos a segunda linha com a terceira:

(15)

Neste passo temos uma matriz da Equação (15) na forma triangular (chamada de matriz escalonada da Equação). Da terceira linha, encontramos -2z = -2, ou seja, z = 1. Substituindo na segunda equação, temos –y - 3z = -2, ou seja, y = -1 e finalmente, da primeira linha, x + y + z = 1, resultando em x = 1.

A.2 – Decomposição LU

Dado um sistema linear a ser resolvido pelo método de decomposição *LU* como:

(20)

Começamos fatorando a matriz *A* dos coeficientes da a Equação (20):

(21)

(22)

Finalizada a decomposição *LU* da Equação (21), resolvemos primeiramente a Equação *Ly = b,* através do resultado da Equação (22)*:*

(23)

Onde através do resultado da Equação (23) obtemos = -2,  = 5 e = -8. Assim obtemos a solução resolvendo a Equação *Ux = y:*

(24)

Resultando na resolução do sistema linear em = -1,  = -2 e  = 1 através do resultado da Equação (24).

A.3 – QR

Dado um sistema linear a ser resolvido pelo método de decomposição *QR* usando uma matriz inicial *A* que como:

(27)

Inicialmente definir a matriz *Q* como a matriz formada pelos vetores desta base ortonormal da Equação (27):

(28)

Vimos que *T* portanto para o resultado da Equação (28):

*T*

(29)

Finalmente encontrar a resolução do sistema linear pela decomposição *QR* da Equação (28) e (29):

(30)

A.4 – Decomposição de Cholesky

Dado um sistema linear a ser resolvido pelo método de decomposição Cholesky usando uma matriz inicial *A* que como:

Temos o produto da matriz triangular (Equação 34) e da sua transposta :

*A*  =

Note que cada matriz simétrica de definição positiva tem uma decomposição Cholesky única, e todos os valores diagonais de são positivos. O Algoritmo Cholesky é uma versão modificada e recursiva da eliminação gaussiana. A partir do canto superior esquerdo e itera a matriz linha por linha (Equação 35).

A.5 – Gauss Jacobi

Para um sistema linear a ser resolvido pelo método Gauss Jacobi com aproximação inicial (0) = [0, 0] T e τ = 10−4 como critério de parada:

(38)

Na primeira iteração da Equação (38), encontramos:

Com:

Na segunda iteração da Equação (38), encontramos:

Com:

Na terceira iteração da Equação (38), encontramos:

Com:

Procedemos sucessivas iterações da Equação (38), até que na décima nona iteração, encontramos:

Com:

10−5

A.6 – Gauss Seidel

Representação pelo método Gauss Seidel:

(41)

*A* =

Para um sistema linear a ser resolvido pelo método Gauss Seidel com erro < 10−2:

(43)

Verificando a garantia de convergência do método de Gauss Seidel para a Equação (43)**:**

**-** A matriz não é estritamente dominante. **FALHA.**

**-** Pelo critério das linhas *max*1≤i≤n **FALHA.**

**-** Critério de Sassenfeld.

*max*1≤i≤n *βi* < 1

seja *max*1≤i≤n *βi* < 1.

em que *βi =*

*β1* = 0.20 + 0.20 = 0.40

*β2* = |0.75| 0.40 + 0.25 = 0.55

*β3* = |0.50| 0.40 + |0.50| 0.55 = 0.475

*max*{0.40, 0.55, 0.475} = 0.55 < 1

Logo, segundo este critério o método de Gauss Seidel para a Equação (43) converge.

A partir de x(0) = (0, 0, 0)T, temos x(1):

O erro relativo é igual a:

A partir de x(1) = (, , )T, temos x(2):

O erro relativo é igual a:

Desta forma as iterações serão processadas até que o critério do erro relativo seja satisfeito para a Equação (43).

A.7 – SVD

Valores e vetores próprios para matrizes simétricas (SVD)

Seja a matriz:

Onde: Auto-valores: λ1 = 4 e λ2 = 1

Auto-vetores: V1 = [0 1]T e V2 = [1 0] T

Valores Singulares: σ1 = 2 e σ2 = 1

A matriz ∑ é:

V é a matriz ortogonal *n* x *n*, portanto do tipo ATA, no caso presente, 2x2. Para construir V basta encontrar o conjunto de vetores fruto de ATA:

# Apêndice B – Bibliotecas

B.1 – Status da Biblioteca cuSolverDN

Vários identificadores CUSOLVERcom configurações diferentes podem ser criados invocando um ponteiro.

**Tabela 6:** Parâmetros de entrada do *cuSolverDN*

|  |  |  |  |
| --- | --- | --- | --- |
| Parâmetro | Memória | Entrada/Saída | Significado |
| handle | host | output | O ponteiro para identificar o contexto *cuSolverDN*. |

**Fonte:** (NVIDIA, *cuSOLVER library*, 2019).

O *cusolverCreate* aloca alguns recursos internos para um determinado dispositivo como visto na Tabela 6.

**Tabela 7:** Status retornado

|  |  |
| --- | --- |
| Parâmetro | Significado |
| CUSOLVER\_STATUS\_SUCCESS | A inicialização foi bem sucedida. |
| CUSOLVER\_STATUS\_INVALID\_VALUE | Parâmetros inválidos foram passados ​​(*m, n* < 0 ou lda < max (1, *m*)) |
| CUSOLVER\_STATUS\_NOT\_INITIALIZED | A inicialização do CUDA *Runtime* falhou. |
| CUSOLVER\_STATUS\_ALLOC\_FAILED | Os recursos não puderam ser alocados. |
| CUSOLVER\_STATUS\_ARCH\_MISMATC | O dispositivo suporta apenas a capacidade de computação 2.0 ou superior. |
| CUSOLVER\_STATUS\_INTERNAL\_ERRO | Uma operação interna falhou. |

**Fonte:** (NVIDIA, *cuSOLVER library*, 2019)

Na tabela 7 tem-se os principais retornos de status esperados ao alocar um dispositivo do identificador *CUSOLVER*.

B.2 – cuBLAS (Eliminação Gaussiana)

Vários identificadores CUBLAScom configurações diferentes podem ser criados.

**Tabela 8:** Parâmetros de entrada do *cuBlas*

|  |  |  |  |
| --- | --- | --- | --- |
| Parâmetro | Memória | Entrada/Saída | Significado |
| handle | host | output | O ponteiro para identificar o contexto *cuBlas*. |

**Fonte:** (NVIDIA, *cuBlas library*, 2019)

O *cublasCreate* aloca alguns recursos internos para um determinado dispositivo como visto na Tabela 8, vários identificadores *CUBLAS* com configurações diferentes podem ser criados.

**Tabela 9:** Status retornado

|  |  |
| --- | --- |
| Parâmetro | Significado |
| CUBLAS\_STATUS\_SUCCESS | A inicialização foi bem sucedida |
| CUBLAS\_STATUS\_NOT\_INITIALIZED | A inicialização do CUDA ™ Runtime falhou |
| CUBLAS\_STATUS\_ALLOC\_FAILED | Os recursos não puderam ser alocados |

**Fonte:** Status retornado (NVIDIA, *cuBlas library*, 2019).

Na tabela 9 tem-se os principais retornos de status esperados ao alocar um dispositivo do identificador *CUBLAS*.

**Tabela 10:** Parâmetros de entrada do *cublasZgemm3m*

|  |  |  |  |
| --- | --- | --- | --- |
| Parâmetro | Memória | Entrada/Saída | Significado |
| handle |  | input | Identificador para o contexto da biblioteca *cuBLAS.* |
| transa |  | input | Operação *A* que não é transposta. |
| transb |  | input | Operação *B* que não é transposta |
| m |  | input | Número de linhas da matriz *A* e *C*. |
| n |  | input | Número de colunas da matriz *B* e *C.* |
| k |  | input | Número de colunas da matriz *A* e linhas de *B*. |
| alpha | host or device | input | Escalar usado para multiplicação. |
| A | device | input | Matriz de dimensões lda x k com lda > = max (1, *m*) se transa == CUBLAS\_OP\_N e lda x m com lda > = max (1, *k*) caso contrário |
| lda |  | input | Dimensão principal da matriz bidimensional usada para armazenar a matriz *A*. |
| B | device | input | Matriz da dimensão ldb x n com ldb > = max (1, k) se transb == CUBLAS\_OP\_N e ldb x k com ldb > = max (1, n) caso contrário. |
| ldb |  | input | Dimensão principal da matriz bidimensional usada para armazenar a matriz *B*. |
| beta | host or device | input | Escalar usado para multiplicação. Se beta = 0, C não precisa ser uma entrada válida. |
| C | device | in/out | Matriz de dimensões ldc x n com ldc > = max (1, *m*). |
| ldc |  | input | Dimensão principal de uma matriz bidimensional usada para armazenar a matriz C. |

**Fonte:** (NVIDIA, *cuBlas library*, 2019)

Na tabela 10 tem-se os principais parâmetros para a biblioteca *cublasCgemm3m.*

B.3 – cuSolverDN (Decomposição LU)

Vários identificadores CUSOLVERcom configurações diferentes podem ser criados.

**Tabela 11:** Parâmetros de entrada do *cuSolverDnSgetrf e cuSolverDnSgetrs*

|  |  |  |  |
| --- | --- | --- | --- |
| Parâmetro | Memória | Entrada/Saída | Significado |
| handle | host | input | O ponteiro para identificar o contexto *cuSolverDN*. |
| trans | host | input | Operação op(*A*) que não é ou (trans.) transpõe. |
| m | host | input | Número de linhas da matriz *A*. |
| n | host | input | Número de colunas da matriz *A*. |
| nrhs | host | input | Número de lados do lado direito. |
| A | device | in/out | <type> array de dimensão lda \* *n.* Com lda não é menor que max (1, *m*). |
| B | device | output | <type> array de dimensão ldb \* nrhs com ldb não é menor que max (1, *n*). |
| lda | host | input | Dimensão principal da matriz bidimensional usada para armazenar a matriz *A*. |
| ldb | host | input | Dimensão principal da matriz bidimensional usada para armazenar a matriz *B*. |
| workspace | device | in/out | Espaço de trabalho, <type> matriz de tamanho *Lwork.* |
| devIpiv | device | output | Matriz de tamanho min (*m, n*), contendo índices de pivô. |
| devInfo | device | output | Se devInfo = 0, a decomposição LU será bem-sucedida. Se devInfo = -i, o i-ésimo parâmetro está errado (sem contar o identificador). Se devInfo = i, o U (i, i) = 0. |

**Fonte:** (NVIDIA, *cuSOLVER library*, 2019)

Na tabela 11 tem-se os principais parâmetros para a biblioteca *cusolverDnSgetrf* e *cusolverDnSgetrs.*

B.4 – cuSolverDN (Decomposição QR)

Vários identificadores *CUSOLVER* com configurações diferentes podem ser criados.

**Tabela 12:** Parâmetros de entrada para *cuSolverDnDgeqrf e cuSolverDnDorgqr*

|  |  |  |  |
| --- | --- | --- | --- |
| Parâmetro | Memória | Entrada/Saída | Significado |
| handle | host | input | Identificador para o contexto da biblioteca *cuSolverDN*. |
| m | host | input | número de linhas da matriz *A.* |
| n | host | input | Número de colunas da matriz *A*. |
| A | device | in/out | <type> matriz da dimensão lda \* *n* com lda não é menor que max (1*, m*) |
| lda | host | input | Dimensão principal da matriz bidimensional usada para armazenar a matriz *A*. |
| TAU | device | output | <type> Matriz de dimensão pelo menos min(*m, n).* |
| workspace | device | in/out | Espaço de trabalho, <type> matriz de tamanho *Lwork.* |
| Lwork | host | input | tamanho da matriz de trabalho. |
| devInfo | device | output | Se devInfo = 0, a decomposição será bem-sucedida. se devInfo = -i, o i-ésimo parâmetro está errado (sem contar o identificador). |

**Fonte:** (NVIDIA, *cuSOLVER library*, 2019)

Na tabela 12 temos os principais parâmetros para a biblioteca *cusolverDnDgeqrf e cusolverDnDorgqr*.

B.5 – cuSolverDN (Decomposição de Cholesky)

Vários identificadores *CUSOLVER* com configurações diferentes podem ser criados.

**Tabela 13:** Parâmetros de entrada do *cuSolverDnSpotrf* e *cuSolverDnSpotrs*

|  |  |  |  |
| --- | --- | --- | --- |
| Parâmetro | Memória | Entrada/Saída | Significado |
| handle | host | Input | Identificador para o contexto da biblioteca cuSolverDN. |
| uplo | host | Input | Indica se a parte inferior ou superior da matriz *A* está armazenada, a outra parte não é referenciada. |
| n | host | input | Número de linhas e colunas da matriz *A*. |
| nrhs | host | input | Número de colunas da matriz *X* e *B*. |
| A | device | input | <type> matriz da dimensão lda \* *n* com lda não é menor que max(1*, n*). |
| B | device | in/out | <type> Matriz de dimensão ldb \* nrhs. ldb não é menor que max (1, n). Como entrada, *B* é a matriz do lado direito. Como saída, *B* é a matriz da solução. |
| lda | host | input | Dimensão principal da matriz bidimensional usada para armazenar a matriz *A.* |
| Workspace | device | in/out | Espaço de trabalho, <type> matriz de tamanho *Lwork* |
| Lwork | device | in/out | Tamanho do espaço de trabalho, retornado por *potrf\_bufferSize*. |
| devInfo | device | output | Se devInfo = 0, a decomposição de Cholesky é bem-sucedida. se devInfo = -i, o i-ésimo parâmetro está errado (sem contar o identificador). se devInfo = i, o menor menor da ordem i não é positivo definitivo. |

**Fonte:** (NVIDIA, *cuSOLVER library*, 2019)

Na tabela 13 temos os principais parâmetros para as bibliotecas *cusolverDnSpotrf* e *cusolverDnSpotrs*.

**Tabela 14:** Parâmetros de entrada do *cusolverDnSsyevj* e *cusolverDnDsyevj*

|  |  |  |  |
| --- | --- | --- | --- |
| Parâmetro | Memória | Entrada/Saída | Significado |
| handle | host | input | Identificador para o contexto da biblioteca cuSolverDN. |
| jobz | host | input | Especifica opções para calcular apenas o autovalor ou o par de autovalores:  jobz = *CUSOLVER\_EIG\_MODE\_NOVECTOR*: Calcular apenas autovalores;  jobz = *CUSOLVER\_EIG\_MODE\_VECTOR*: Calcular valores próprios e vetores próprios. |
| uplo | host | input | Especifica qual parte de A está armazenada.  uplo = *CUBLAS\_FILL\_MODE\_LOWER*: O triângulo inferior de *A* é armazenado.  uplo = *CUBLAS\_FILL\_MODE\_UPPER*: O triângulo superior de *A* é armazenado. |
| n | host | input | Número de linhas (ou colunas) da matriz *A*. |
| A | device | in/out | <type> Array de dimensão lda \* n com lda não é menor que max (1, *n*).  Se uplo = *CUBLAS\_FILL\_MODE\_UPPER*, a parte triangular superior n-por-n principal de *A* contém a parte triangular superior da matriz *A*.  Se uplo = *CUBLAS\_FILL\_MODE\_LOWER*, a parte triangular inferior n-por-n principal de *A* contém a parte triangular inferior de a matriz *A*.  Na saída, Se jobz = *CUSOLVER\_EIG\_MODE\_VECTOR* e info = 0, A contém os autovetores ortonormais da matriz *A.* Se jobz = *CUSOLVER\_EIG\_MODE\_NOVECTOR,* o conteúdo de *A* é destruído. |
| lda | host | input | Dimensão principal da matriz bidimensional usada para armazenar a matriz *A*. |
| W | device | output | Uma matriz real de dimensão *n*. Os valores do autovalor de *A*, em ordem crescente, ou seja, classificados de forma que W (i) <= W (i + 1). |
| work | device | in/out | Espaço de trabalho, <tipo> matriz de tamanho de trabalho. |
| lwork | host | input | Tamanho do trabalho, retornado por *syevj\_bufferSize.* |
| info | device | output | Se info = 0, a operação foi bem-sucedida.  Se info = -i, o i-ésimo parâmetro está errado (sem contar o identificador).  Se info = n + 1, a dose *syevj* não converge sob a tolerância dada e as varreduras máximas. |
| params | host | in/out | Estrutura preenchida com parâmetros do algoritmo de Jacobi e resultados do *syevj.* |

**Fonte:** (NVIDIA, *cuSOLVER library*, 2019)

Na tabela 14 temos os principais parâmetros para as bibliotecas *cusolverDnSsyevj* e *cusolverDnDsyevj*.

B.6 – cuSparce (Gauss Seidel)

Vários identificadores CUSPARCEcom configurações diferentes podem ser criados.

**Tabela 15:** Parâmetros de entrada do *cuSparce*

|  |  |  |  |
| --- | --- | --- | --- |
| Parâmetro | Memória | Entrada  Saída | Significado |
| handle | host | input | Identificador para o contexto da biblioteca cuSPARSE. |
| trans |  |  | Transposta da matriz *A* |
| m |  |  | Número de linhas e colunas da matriz *A* |
| alpha |  |  | <type> Escalar usado para multiplicação. |
| descrA |  |  | O descritor da matriz A. Os tipos de matriz suportados são *CUSPARSE\_MATRIX\_TYPE\_TRIANGULAR* e *CUSPARSE\_MATRIX\_TYPE\_GENERAL*, enquanto os tipos diagonais suportados são *CUSPARSE\_DIAG\_TYPE\_UNIT* e *CUSPARSE\_DIAG\_TYPE\_NON\_UNIT.* |
| csrSortedValA |  |  | <type> Matriz de nnz (= csrSortedRowPtrA (m) - csrSortedRowPtrA (0)) elementos diferentes de zero da matriz *A*. |
| csrSortedRowPtrA |  |  | Matriz inteira de *m* + 1 elementos que contém o início de cada linha e o final da última linha mais um. |
| csrSortedColIndA |  |  | Matriz inteira de índices de coluna nnz (= csrSortedRowPtrA (*m*) - csrSortedRowPtrA (0)) dos elementos diferentes de zero da matriz *A*. |
| info |  |  | Estrutura com informações coletadas durante a fase de análise (que deveriam ter sido passadas para a fase de solução inalterada). |
| F |  |  | <type> Vetor do lado direito do tamanho *m.* |
| x |  | output | <type> Vetor de solução de tamanho *m.* |

**Fonte:** (NVIDIA, *cuSPARCE library*, 2019)

Na tabela 15 temos os principais parâmetros para as bibliotecas *cusparseDcsrsv\_solve*.

B.7 – cuSolverDN (SVD)

Vários identificadores *CUSOLVER* com configurações diferentes podem ser criados.

**Tabela 16:** Parâmetros de entrada do *cuSolverDnSsyevd*

|  |  |  |  |
| --- | --- | --- | --- |
| Parâmetro | Memória | Entrada/Saída | Significado |
| handle | host | input | Identificador para o contexto da biblioteca *cuSolverDN*. |
| jobz | host | input | Especifica opções para calcular apenas o autovalor ou o par de autovalores:  jobz = *CUSOLVER\_EIG\_MODE\_NOVECTOR*: Calcular apenas autovalores;  jobz = *CUSOLVER\_EIG\_MODE\_VECTOR*: Calcule valores próprios e vetores próprios. |
| uplo | host | input | Especifica qual parte de *A* está armazenada.  uplo = CUBLAS\_FILL\_MODE\_LOWER: O triângulo inferior de A é armazenado.  uplo = CUBLAS\_FILL\_MODE\_UPPER: O triângulo superior de A é armazenado. |
| n | host | input | Número de linhas (ou colunas) da matriz *A*. |
| A | device | in/out | <type> Array de dimensão lda \* n com lda não é menor que max (1, *n*). Se uplo = *CUBLAS\_FILL\_MODE\_UPPER*, a parte triangular superior n-por-n principal de *A* contém a parte triangular superior da matriz *A*. Se uplo = *CUBLAS\_FILL\_MODE\_LOWER*, a parte triangular inferior n-por-n principal de *A* contém a parte triangular inferior de a matriz A. Na saída, se jobz = CUSOLVER\_EIG\_MODE\_VECTOR e devInfo = 0, *A* contém os autovetores ortonormais da matriz *A*. Se jobz = CUSOLVER\_EIG\_MODE\_NOVECTOR, o conteúdo de *A* é destruído. |
| lda | host | input | Dimensão principal da matriz bidimensional usada para armazenar a matriz *A*. |
| W | device | output | Uma matriz real de dimensão *n*. Os valores do autovalor de *A*, em ordem crescente, ou seja, classificados de forma que W (i) <= W (i + 1). |
| work | device | in/out | Espaço de trabalho, <tipo> matriz de tamanho de trabalho. |
| Lwork | host | input | Tamanho do trabalho, retornado por *syevd\_bufferSize.* |
| devInfo | device | output | Se devInfo = 0, a operação será bem sucedida. se devInfo = -i, o i-ésimo parâmetro está errado (sem contar o identificador). se devInfo = i (> 0), devInfo indica i elementos fora da diagonal de uma forma tridiagonal intermediária não convergiram para zero; |

**Fonte:** (NVIDIA, *cuSOLVER library*, 2019)

Na tabela 16 temos os principais parâmetros para a biblioteca *cusolverDnSsyevd*.

# Apêndice C – Algoritmo proposto com ladrilhamento

#!/bin/bash

trap "exit" INT

pathExe="/usr/local/cuda/samples/bin/x86\_64/linux/release"

pathBases=$PWD/asbases

echo "QR"

for entry in `ls $PWD/asbases`; do

for counter in {1..2}; do

$pathExe/./cuSolverSp\_LinearSolver -R=qr -file=$pathBases/$entry > temp\_$entry

cat temp\_$entry | grep -i "timing" | tr -s ' ' | cut -d ' ' -f 10 >> times\_gpu\_qr\_$entry.txt

cat temp\_$entry | grep -i "timing" | tr -s ' ' | cut -d ' ' -f 5 >> times\_cpu\_qr\_$entry.txt

# $pathExe/./cuSolverSp\_LinearSolver -R=qr -file=$pathBases/$entry | grep -i "timing" | tr -s ' ' | cut -d ' ' -f 5 >> times\_cpu\_qr\_$entry.txt

done

echo $entry

echo "GPU : "

awk '{ sum += $0 } END { print sum/NR }' times\_gpu\_qr\_$entry.txt

echo "CPU : "

awk '{ sum += $0 } END { print sum/NR }' times\_cpu\_qr\_$entry.txt

done

echo "\n"

echo "LU"

for entry in `ls $PWD/asbases`; do

for counter in {1..2}; do

$pathExe/./cuSolverSp\_LinearSolver -R=lu -file=$pathBases/$entry > temp\_$entry

#$pathExe/./cuSolverSp\_LinearSolver -R=lu -file=$pathBases/$entry | grep -i "timing" | tr -s ' ' | cut -d ' ' -f 10 >> times\_gpu\_lu\_$entry.txt

cat temp\_$entry | grep -i "timing" | tr -s ' ' | cut -d ' ' -f 10 >> times\_gpu\_lu\_$entry.txt

cat temp\_$entry | grep -i "timing" | tr -s ' ' | cut -d ' ' -f 5 >> times\_cpu\_lu\_$entry.txt

#$pathExe/./cuSolverSp\_LinearSolver -R=lu -file=$pathBases/$entry | grep -i "timing" | tr -s ' ' | cut -d ' ' -f 5 >> times\_cpu\_lu\_$entry.txt

done

echo $entry

echo "GPU :"

awk '{ sum += $0 } END {print sum/NR }' times\_gpu\_lu\_$entry.txt

echo "CPU :"

awk '{ sum += $0 } END {print sum/NR }' times\_cpu\_lu\_$entry.txt

done

echo "\n"

echo "CHOL"

for entry in `ls $PWD/asbases`; do

nameMatrix=`ls $PWD/asbases/$entry | grep \*.mtx`

for counter in {1..2}; do

$pathExe/./cuSolverSp\_LinearSolver -R=chol -file=$pathBases/$entry > temp\_$entry

cat temp\_$entry | grep -i "timing" | tr -s ' ' | cut -d ' ' -f 10 >> times\_gpu\_chol\_$entry.txt

cat temp\_$entry | grep -i "timing" | tr -s ' ' | cut -d ' ' -f 5 >> times\_cpu\_chol\_$entry.txt

# /$pathExe/./cuSolverSp\_LinearSolver -R=chol -file=$pathBases/$entry | grep -i "timing" | tr -s ' ' | cut -d ' ' -f 10 >> times\_gpu\_chol\_$entry.txt

# /$pathExe/./cuSolverSp\_LinearSolver -R=chol -file=$pathBases/$entry | grep -i "timing" | tr -s ' ' | cut -d ' ' -f 5 >> times\_cpu\_chol\_$entry.txt

done

echo $entry

echo "GPU :"

awk '{ sum += $0 } END {sum/NR }' times\_gpu\_chol\_$entry.txt

echo "CPU :"

awk '{ sum += $0 } END {sum/NR }' times\_cpu\_chol\_$entry.txt

done