Marcelo Brandão Monteiro dos Santos

Aceleração do Cálculo de Autovalores Usando CUDA: Uma Aplicação em Heteroestruturas Semicondutoras

Brasília

Novembro de 2014

Ficha catalográfica elaborada pela Biblioteca Central da Universidade de Brasília. Acervo 1019012.

Santos, Marcelo Brandão Monteiro dos.
S237a Aceleração do cálculo de autovalores usando CUDA : uma aplicação em heteroestruturas semicondutoras / Marcelo Brandão Monteiro dos Santos. -- 2014.
62 f. : il. ; 30 cm.

Dissertação (mestrado) - Universidade de Brasília, Faculdade de Planaltina, Programa de Pós-Graduação em Ciências de Materiais, 2014.
Orientação: Bernhard Georg Enders Neto.
Inclui bibliografia.

1. Autovalores. 2. Processamento paralelo (Computadores). 3. Schrodinger, Equação de. 4. Microprocessadores. 5. Semicondutores. I. Enders Neto, Bernhard Georg. II. Título.

CDU 620.1

Marcelo Brandão Monteiro dos Santos

Aceleração do Cálculo de Autovalores Usando CUDA: Uma Aplicação em Heteroestruturas Semicondutoras

Dissertação apresentada ao programa de Pós-Graduação em Ciências de Materiais da Universidade de Brasília, campus Planaltina, como requisito parcial para a obtenção do título de mestre em Ciências de Materiais.

Orientador:
Bernhard Georg Enders Neto

Universidade de Brasília

Brasília

Novembro de 2014

Aceleração do Cálculo de Autovalores Usando CUDA: Uma Aplicação em Heteroestruturas Semicondutoras

por

Marcelo Brandão Monteiro dos Santos

Campus Planaltina
Universidade de Brasília

Dissertação apresentada ao programa de Pós-Graduação em Ciências de Materiais da Universidade de Brasília, campus Planaltina, como requisito parcial para a obtenção do título de mestre em Ciências de Materiais. Aprovada em 08 de novembro de 2014 pela seguinte banca examinadora:

Bernhard Georg Enders Neto
Orientador – Universidade de Brasília

Rafael Morgado Silva
Examinador – Universidade de Brasília

Paulo Eduardo de Brito
Examinador – Universidade de Brasília

Aos meus Pais. Aos amigos.

A vela se apaga,

o olhar paira por mais que um minuto na escuridão,

lhe vem um pensando, de um menino, Ernest...

"Somo aprendizes de uma arte na qual ninguém se torna mestre."

não nos tornaremos mestres, não

Pegue outra vela, acenda,

não seremos mestres, mas seremos, eternos curiosos

meninos, curiosos, apenas isso.

(Marcelo Brandão)

Resumo

Inicialmente projetadas para processamento de gráficos, as placas gráficas (GPUs) evoluíram para processadores paralelos de propósito geral de alto desempenho. Usando unidades de processamento gráfico (GPUs), da NVIDIA, adaptamos métodos (algoritmos) computacionais de linguagem C para linguagem CUDA. Resolvemos a equação de Schrödinger pelo método de diferenças finitas, usando o método da Bissecção com sequência de Sturm para um poço quântico simétrico de heteroestruturas de GaAs/AlGaAs com a finalidade de acelerar a busca do autovalores. Comparamos o tempo gasto entre os algoritmos usando a GPU, a CPU e a rotina DSTEBZ da biblioteca Lapack. Dividimos o problema em duas fases, a de isolamento, calculada na CPU, e a de extração, calculada na GPU, na fase de extração o método em GPU foi cerca de quatro vezes mais rápido que o método na CPU. O método híbrido, isolamento na CPU e extração na GPU foi cerca de quarenta e seis vezes mais rápido que a rotina DSTEBZ.

Abstract

Initially designed for graphics processing, the (GPU) graphics cards have evolved into general purpose parallel processors for high performance. Using graphics processing units (GPUs), NVIDIA, adapt computing methods (algorithms) C language for CUDA language. We solve the Schrödinger equation by the finite difference method, using the Bisection method with Sturm sequence for a symmetric quantum well heterostructures of GaAs / AlGaAs. In order to accelerate the search for eigenvalues. We compared the time spent between algorithms using the GPU, CPU and DSTEBZ routine LAPACK library. The problem divided into two phases, the insulation calculated in the CPU and extracting calculated in the GPU, in phase extraction method GPU was about four times faster than the method in the CPU. The hybrid method, isolating on the CPU and extraction on the GPU was about forty-six times faster than DSTEBZ routine.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Abreviaturas e Siglas

CPU – Unidade central de processamento

AMD – Advanced Micro Devices

CUDA – Compute Unified Device Architecture

Ga – Gálio

Al – Alumínio

GaAs – Arseneto de gálio

$Al_xGa_{1-x}As$ – Arseneto de gálio-alumínio

EQSCD – equação de Schrödinger

GPU – unidade de processamento gráfico

GPGPU – General-Purpose computing on Graphics Processing Units

SMs – Streaming Multiprocessors

MKL – Intel® Math Kernel Library

ED – Equaçao diferencial

EDF – Equaçao diferencial discretizada

ELT – Erro local de truncamento

LAPACK – Linear algebra package

MGPU – Múltiplas unidades de processamento gráfico

MCPU – Múltiplas unidades de processamento central

1 Introdução

Desde 1970, quando a Intel desenvolve a CPU 4004, com incríveis 8 Mhz de velocidade na época, a indústria de processadores veio ganhando corpo e trabalhando para construir núcleos cada vez mais velozes, a AMD, Advanced Micro Devices entrou no nesse mercado em 1982, e desde então esta são as empresas que ao longo do tempo passaram a dominar o mercado.

Com as constantes evoluções no campo de pesquisas, foi possível até o ano de 2003, produzir CPUs cada vez mais rápidos, sendo que os microprocessadores chegam a ser capazes de fazer bilhões de operações de ponto flutuante por segundo (GFLOPS) e em clusters, chega-se as casa dos cem milhões de GFLOPS de operações por segundo.

Contudo, em 2003, o ganho de processamento ao se produzir uma CPU mais rápida passou a não ser vantajoso, devido ao auto gasto energético e a dissipação do calor gerado, foi então, que a indústria de microprocessadores optou por construir microprocessadores com mais núcleos ao invés de aumentar a capacidade de um único núcleo.

Tendo tomado a decisão de mudar o número de núcleos em cada processador, a indústria de microprocessadores gerou um problema que atingiu ela e a indústria de desenvolvimento de software, pois os programas eram feitos para serem lidos de forma sequêncial em um núcleo, com a introdução de mais núcleos, a forma como os programas são pensados e executados deveria sofrer alterações ou simplesmente não utilizar toda a capacidade que os microprocessadores poderiam oferecer, como consequência teríamos processadores mais potentes, mas com essa potência desperdiçada.

Pensando nisso, a indústria de processadores, fez com que os programas sequênciais fosse executados movendo-os entre os múltiplos núcleos, conceito

que é conhecido como *multicore trajectory*, e mais tarde a Intel criou a tecnologia *hyperthreading*, na qual o microprocessador simula ter mais núcleos, e maximiza a velocidade de execução dos programas sequenciais, em contrapartida, foi pensada também uma forma de fazer os programas rodarem em vários núcleos ao mesmo tempo, conceito que ficou conhecido como *many-core trajectory*.

Uma das principais vantagens em usar as placas gráficas, GPUs, é que as threads das GPUs são muito mais leves do que as threads dos processadores, CPUs, uma forma de comparação é pensar na thread de GPU leve como uma galinha e a de CPU pesada como um boi. Então o gasto computacional para dispara uma thread de GPU é muito menor que o para dispara uma thread de CPU.

Como exemplo podemos citar a Nvidia GeForce GTX 280 que trabalha com 240 núcleos cada um com *multithread*. As tecnologias *many-core*, em especial as unidades de processamento gráfico (GPUs) tem liderado a corrida e mostrando grande desempenho ao resolver cálculos em ponto flutuante, conforme ilustrado na figura 1. [1]

Em 2007, a NVIDIA, lançou oficialmente o paradigma de programação CUDA, que é um modelo geral de programação em paralelo para o uso em GPUs, disponível a partir de suas placas GeForce série 8. Estas placas são formadas por várias unidades de processamentos, com memória compartilhada e capazes de executar milhares de *threads*.

Já em 2008, um ano após o lançamento oficial do CUDA, a Nvidia aparece na lista TOP500, com o TSUBAME, supercomputador construído em parceria com a Tokyo Institute of Technology, o qual possuia 170 GPUs Tesla S1070, ocupando a 29 posição no rank, já na lista publicada em novembro de 2012, temos o supercomputador *TITAN* ocupando a primeira posição, o qual possui 18.688 nós, cada um com um processador de 16 núcleos AMD Opteron 6274 e um placa gráfica NVIDIA tesla K20 que possuem 2496 núcleos de processamento cada, as placa gráficas, segundo a NVIDIA, reponde por cerca de 90% do processamento do Titan. O Titan é usado para fornecer uma computação sem precedentes para as pesquisa de mudança climática, motores mais eficientes, materiais e outros

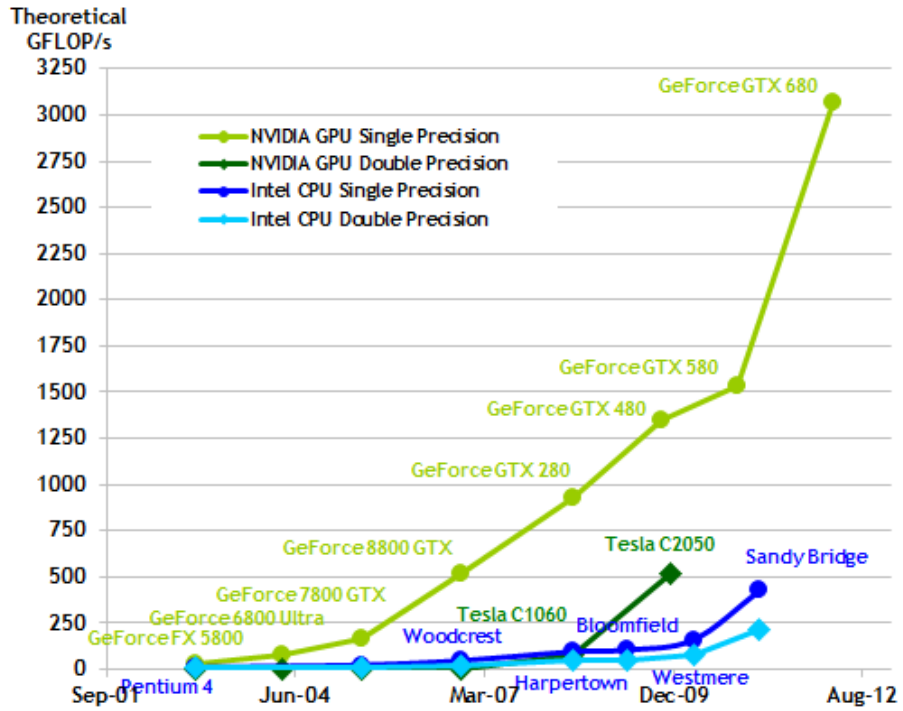

FIGURA 1 – Comparação entres as GPUs e CPUs

Fonte: NVIDIA [2]

temas. Na mais atual do top500, a de novembro de 2013, o *TITAN* configura como o segundo computador mais rápido do mundo, perdendo apenas para o *Tianhe-2 (MilkyWay-2)* do National Super Computer Center em Guangzhou, na China.

O supercomputador com a maior performance em solo brasileiro, que consta da lista é o Grifo04, na posição 98 na lista TOP500, foi produzido pela Itautec para a Petrobrás e é usado para o estudos de processamento sísmico, que serão importantes na busca de petróleo na camada pré-sal, na sua construção foram utilizadas 1088 placas gráficas NVIDIA tesla M2050, tendo uma performance medida pelo *benchmark linpack* de 251,5 TFlops/s.

Analisando esses pontos temos que as GPUs mostram um caminho a ser seguido na computação de alta performance, em relação ao custo e a capacidade de paralelizar dos programas. Comparativamente em 2009 já tínhamos GPUs trabalhando a 1 teraflops em precisão simples enquanto os processadores Multicore estavam na casa de 100 gigaflops em precisão dupla.

1.1 Heteroestruturas

Heteroestruturas podem ser definidas como estruturas semicondutoras heterogêneas feitas de dois ou mais semicondutores diferentes [3], de tal forma que a região de transição ou interface dos diferentes materiais desempenha um papel essencial na ação de qualquer nanodispositivo. Os semicondutores participantes das heteroestruturas envolvem os elementos da parte central da tabela periódica (veja Tabela 1 na página seguinte). Cada elemento da coluna III pode ser combinado com um elemento da coluna V para formar o chamado composto III–V. Dos elementos mostrados, doze compostos III–V distintos podem ser formados. O composto mais utilizado é o arseneto de gálio (GaAs), mas todos eles podem ser utilizados em heteroestruturas, a escolha depende da aplicação. Na prática, compostos III–V são quase sempre utilizados em heteroestruturas, ao invés de isolados. Dois ou mais compostos distintos podem ser usados para formar ligas. Um exemplo comum é o arseneto de gálio-alumínio ($Al_xGa_{1-x}As$), onde x é a fração do cristal ocupada por átomos de alumínio (Al) é $1-x$ e a fração ocupada por átomos de gálio (Ga). Assim não temos apenas uma quantidade finita de compostos diferentes, mas uma faixa contínua de materiais, já que é possível fazer heteroestruturas de composição gradual, nas quais a composição varia continuamente ao longo da estrutura do dispositivo.

Durante muito tempo as principais heteroestruturas utilizadas na fabricação de dispositivos eletrônicos e optoeletrônicos de baixa dimensionalidade foram aquelas compostas pelo GaAs e suas ligas. Esse fato pode ser explicado devido a uma gama de vantagens oferecidas por esses materiais, dentre elas podemos citar: ótima qualidade de interface, pequena diferença entre as constantes dielétricas das camadas constituintes, baixa concentração de impurezas residuais, alta mobilidade eletrônica a baixa temperatura, possibilidade de alta concentração de dopagem nas barreiras, dentre outras[4].

Tabela 1 – Porção central da tabela periódica mostrando os elementos das colunas II a VI, utilizados em heteroestruturas.

II	III	IV	V	VI
			N	
	Al	Si	P	S
Zn	Ga	Ge	As	Se
Cd	In		Sb	Te
Hg				

1.2 Equação de Schrödinger

A equação de Schrödinger EQSCD é de suma importância em sistemas quanto-mecânicos, ela desempenha papel análogo, na Mecânica Quântica, ao desempenhado na Física Clássica pela segunda lei de Newton. Resolver a EQSCD significa conhecer a função de onda e os autovalores de energia de um sistema e extrair, a partir daí, valiosas informações associadas ao sistema. Na Figura 2 na próxima página temos o poço de pontecial retangula com os valores dos autovalores encontrados para a heteroestrutura estudada.

A resolução da EQSCD resulta, em geral, em um problema de autovalor que pode ser resolvido por métodos (algoritmos) computacionais. Muitas vezes, contudo, a dimensão do problema de autovalor pode se tornar muito grande de forma a inviabilizar o método numérico utilizado. Também pode acontecer de o tempo gasto na resolução da EQSCD ser grande demais para aplicações em tempo real. Para contornar tal dificuldade, podemos utilizar a computação de propósito geral em unidades de processamento gráfico GPGPU, do inglês *General-Purpose computing on Graphics Processing Units*.

A GPGPU é a técnica que utiliza uma unidade de processamento gráfico (GPU, Graphics Processing Unit), que geralmente lida apenas com a computação de elementos gráficos, para realizar a computação de aplicações tradicionalmente realizadas pela CPU. Atualmente a GPGPU é utilizada, por exemplo, em aplicativos de auditoria de segurança de sistemas, softwares de visualização e também em computação científica.

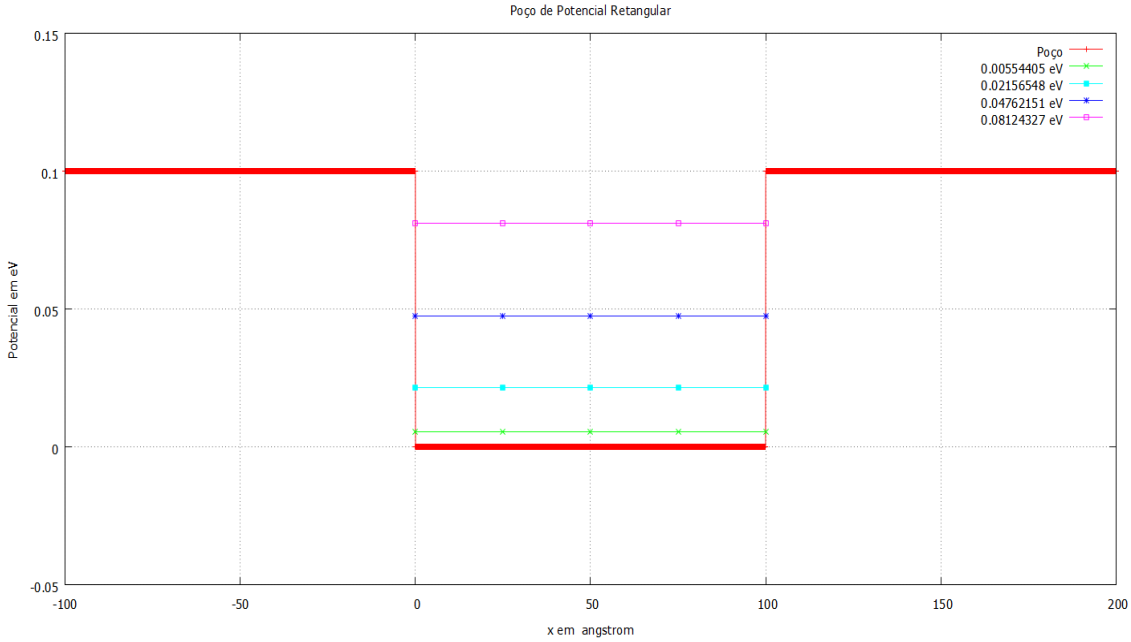

FIGURA 2 – Poço de Pontecial Retangular.

1.3 MOTIVAÇÃO, OBJETIVO E ORGANIZAÇÃO

O objetivo inicial desse trabalho era calcular o Autovalores e os Autovetores da EQSCD, para um poço quântico de potencial retangular usando bibliotecas em C e em CUDA, e analisar o ganho, ou não, de tempo nos cálculos usando o CUDA em relação ao C.

Ao iniciar o trabalho, verificamos que ainda não havia disponíveis bibliotecas em CUDA para o calculo dos Autovalores, havia sim, alguns trabalho sobre os cálculos em matrizes pequenas, como em [5] dentre outros.

Então mudamos o foco e partimos para a implementação de uma rotina para encontrar os autovalores de uma matriz tridiagonal simétrica usando o CUDA. Depois de bastante pesquisa, optamos por montar um algoritmo usando método da Bissecção, Ciclos de Gershgorin e Sequencia de Sturm para isolar e extrair os autovalores.

O trabalho consiste em cinco etapas básicas. A primeira é discretizar a equação de Schroedinger unidimensional para o poço quântico retangular. A segunda é gerar a matriz tridiagonal simétrica a partir desses dados. A terceira é calcular

usando o teorema de Gershgorin o intervalo que contém todos os autovalores. A quarta, chamada posteriormente, fase de isolamento, é subdividir o intervalo que contém todos os autovalores em intervalos que contenham somente um autovalor. A quinta e ultima fase, chamada posteriormente de fase de extração, consiste em extrair ou calcular o autovalor contido em cada intervalo encontrado que foi isolado na fase de isolamento.

No capitulo 2, apresentaremos um histórico de evolução das GPUs e as bibliotecas já existentes.

No capitulo 3, abordaremos a base teórica, o formalismo matemático, a discretização da EQSCD, usando o método de diferenças finitas, o método da bissecção, o teorema dos disco de Gershgorin e a Sequência de Sturm.

No capitulo 4, apresentaremos o resultados, o ganho de tempo entre a nossa rotina e a rotina DSTEBZ da biblioteca LAPACK.

2 Evolução das GPUs

2.1 CUDA

As primeiras GPUs foram projetadas para serem aceleradores gráficos, suportando apenas funções fixas e especificas. Mas a partir do final do anos 90 o hardware tornou-se cada vez mais programável. A NVIDIA lançou sua primeira GPU em 1999. Logo os usos das GPUs foram além daquele dados pelos artistas e desenvolvedores de jogos, pesquisadores viram que poderiam tira proveito da tecnologia e sua excelente performance em cálculos de ponto flutuante, começaram a utilizá-la também, nascia nesse ponto a GPGPU, uma GPU de propósito geral.

No princípio era extremamente complexo programar para a GPU. Tinha que se mapear os cálculos científicos e transpô-los a problemas que pudesse ser representados por triângulos e polígonos. Em 2003, liderados por Ian Buck, uma equipe de pesquisadores anunciou o Brook, que foi o primeiro modelo de programação de ampla adoção a ampliar a linguagem C com construções de paralelismo de dados. Usando conceitos como fluxos, Kernels e operadores de redução, o compilador Brook deixou os códigos mais simples de serem codificados que os códigos existentes e eram sete vezes mais rápidos que os códigos similares existentes.[6]

A NVIDIA sabia que um hardware extremamente rápido tinha que ser combinado a ferramentas intuitivas de software e hardware, e por isso convidou Ian Buck para juntar-se à empresa e começar a desenvolver uma solução para executar o C na GPU de forma fluida. Juntando o software e o hardware, a NVIDIA apresentou a CUDA em 2006, a primeira solução do mundo para computação de propósito geral em GPUs.[6] Sendo assim o CUDA é uma plataforma de computação paralela e um modelo de programação inventados pela NVIDIA.[6]

Atualmente tendo conhecimento da linguagem C, pode-se programar em CUDA com uma certa facilidade, a própria NVIDIA oferece o CUDA Toolkit que inclui um compilador, bibliotecas de álgebra a e ferramentas para depuração. Oferece um abrangente treinamento on-line, há vários seminários via web e livros. Mais de 400 universidades ensinam a programação em CUDA, incluindo centros de excelência, pesquisa e treinamento. Anualmente a NVIDIA organiza a GTC - GPU Technology Conference, com o intuito de promover a o debate, a divulgação, o treinamento através de palestras, painéis de discussão tutoriais e mesas redondas reunindo cientistas e desenvolvedores.

2.2 Arquiteturas CUDA

A grande diferença na capacidade de processamento entre as GPUs e as CUPs, se dá por causa do tipo de arquitetura usado em suas construções.

Nas CPUs temos uma forma otimizada para processamento de códigos sequenciais, com um sofisticado controle lógico a fim de permitir que as instruções de uma *thread* possa ser executada em paralelo ou mesmo fora de ordem, mas mantendo a aparência de uma execução sequencial. Usam grandes memórias *cache* a fim de reduzir a latência no acesso aos dados nas aplicações complexas. Em 2009 já era comum o uso de 4 núcleos de processamento em cada processador para permitir uma boa performance para execução de códigos sequenciais.

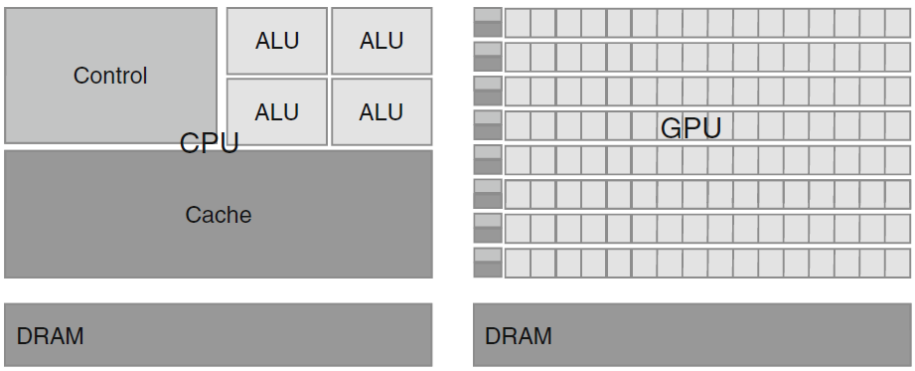

Figura 3 – Arquiteturas das CPUs e GPUs

Fonte: Kirk e Wen-mei [1]

A velocidade de acesso a memória, *memory bandwidth*, influencia de forma significativa na velocidade de processamento, quanto mais rápido pode-se acessar os dados e depois guardá-los, menor será o tempo total de processamento, a *memory bandwidth* da um GPU operam aproximadamente 10 vezes mais rápido que uma CPU contemporânea a essa GPU. [1] Em termos de comparação a GPU GeForce GTX 690 tem uma *memory bandwidth* de 384 GB/sec enquanto os processadores Intel® Core i7-3840QM tem uma taxa de 25,6 GB/s, a GTX 690 tem 3072 núcleos de 915 MHz, já um Core i7 tem 4 núcleos que trabalham a 3,1 MHz.

A forma das GPUs é moldada fortemente pelas indústria de jogos, que exerce forte pressão econômica para que as GPUs tenha capacidade de efetuar um grande numero de operações em ponto-flutuante por segundo por frame de vídeo nos jogos. Essa demanda fez com que os fabricantes maximizassem a área do chip e o uso da energia para as operações em ponto flutuante. Para isso otimizaram a execução de um número massivo de *threads*, com isso algumas procuram cálculos para serem feitos enquanto as outros esperam para acessa a memória, isso minimiza o controle lógico requerido para a execução da *thread*. Pequenas memórias cache ajudam no controle da banda das varias *threads* que acessa a mesma memória sem precisar que todas vão a DRAM, o que acaba por resultar em uma maior área do chip dedicada a cálculos em ponto-flutuante. [1]

A NVIDIA divide suas GPUs em vários *Streaming Multiprocessors*, os SMs, que executam grupos de *threads*, que são chamados de *warps*. Cada um dos SMs possui vários núcleos que são chamados de CUDA cores, núcleos CUDA, cada um possui pipelines completos de operações aritméticas, as Arithmetic Logic Unit, ou ALUs, e de pontos flutuantes, as FPU. Em cada SM há uma memória cache L1 a qual somente os núcleos da respectiva SM tem acesso e todos os núcleos da SM tem acesso à memória global, a L2. A Figura 4 na página seguinte mostra essa organização.

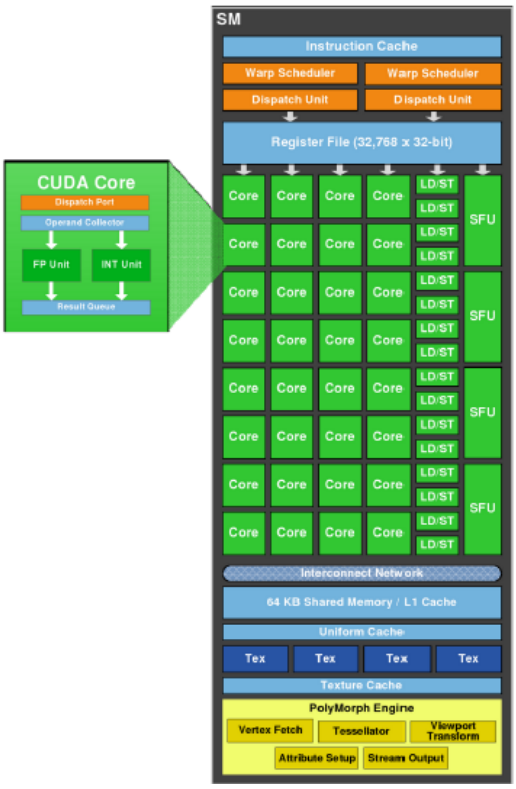

FIGURA 4 – Arquiteturas Fermi

Fonte: Nvidia [7]

2.2.1 *Evolução da Arquitetura das GPUS NVIDIA*

2.2.2 *G80*

Em 2006 a NVIDIA lançou a oitava geração de placas gráficas e a GeForce 8800 foi a primeira GPU com suporte a C e introduzindo a tecnologia CUDA. Foi a primeira a usar uma placa com shaders unificados e 128 núcleos CUDA distribuídos em 8 SMs. [8]

2.2.3 *G200*

Em 2008 foram implementadas novas melhorias CUDA e o suporte a computação de 64-bits de precisão dupla e também houve um aumento no número de núcleos CUDA, passando de 128 para 240.[9]

2.2.4 Fermi

Lançada em abril de 2010, a arquitetura Fermi trouxe suporte ao C++, a alocação dinâmica de objeto e a tratamento de exceções em operações de try e catch. Em um processador Fermi cada SM possui 32 núcleos CUDA e pode executar até 16 operações de precisão dupla por SM em cada ciclo.

2.2.5 Kepler

Em 2012 a NVIDIA lançou a arquitetura Kepler, introduzindo um novo modelo de SM, chamado SMX, que possui 192 núcleos CUDA e um total de 1536 núcleos no chip, que tem 8 SMX. Seu desenvolvimento destacou a importância de obter uma melhor eficiência energética e consequente redução no consumo de energia.

2.3 CUDA na Computação Científica

O CUDA tem sido largamente utilizado em várias aplicações científicas nos últimos anos, agora vamos fazer uma pequena revisão sobre algumas pesquisas que utilizam CUDA.

Dan Negrut, da universidade de Wisconsin-Madison, trabalha com modelos heterogêneos CPU/GPU, exemplos de seus trabalhos são: simulação de terreno granular, estudos de mobilidade de veículos com rodas como tanques, Mars Rover e etc. Analisa as interações fluido-sólidos e analises de elementos finitos não lineares. No artigo *A High Performance Computing Framework for Physics-based Modeling and Simulation of Military Ground Vehicles*, Negrut, Lamb e Gorsich, os últimos dois pesquisadores do exército americanos, demonstra a grande capacidade dos sistemas CPU/GPU ao fazer os cálculos de colisões. São simulados veículos andando sobre a terra, que é vista como um tapete de esferas, portanto o cálculo para determinar as rotas são bastante complexos devido ao grande número de interações entre as rodas e o solo. [10]

Yamazaki e Igarashi do RIKEN Brain Science Institute publicaram o artigo

Realtime cerebellum: A large-scale spiking network model of the cerebellum that runs in realtime using a graphics processing unit, no qual mostra a simulação de um cerebelo, através que um algoritmo de rede neural. O objetivo é que um robô aprenda a bater uma bolinha com um bastão, o modelo desse sistema em GPU foi cerca de 90 vezes mais rápido que o mesmo modelo em CPU. O cerebelo é responsável pelo aprendizado do controle motor, ele continuamente observa o estado das partes do nosso corpo e calcula como devem ser os movimentos adaptando-se. Yamazako e Igarashi usaram para a simulação de uma rede neural, uma GPU GeForce GTX 580, e um robô, o qual com o tempo foi aprendendo o tempo certo de mexer o bastão para acerta a bolinha jogada contra ele, que é um movimento análogo ao de piscar os olhos, um condicionamento Pavloviano.[11] Quando pensamos em um robô, vários sistemas devem estar conectados, o modelo de cerebelo de Yamazako e Igarashi pode constituir um sistema de reflexo para um robô, que poderá ser implementados junto com outros sistemas, navegação, sistema de voz e os necessários ao funcionamento.

Bard at all, no artigo, Cosmological calculations on the GPU, fala sobre a natureza dos cálculos cosmológicos, e grande quantidade de dados que eles envolvem. Cita que a próxima geração de telescópios trará uma grande gama de dados para serem avaliados. As medições cosmológicas necessitam de um grande número de cálculos não triviais. Foram implementados algoritmos para CPU e GPU, e testados com o mesmo número de estrelas. Para calcular a distância angular entre duas galáxias, usando o mesmo método na CPU e na GPU, os equipamentos utilizados foram um processador intel Xeon 2.53 GHz e uma GPU NVIDIA Tesla, com 1000 galáxias, a GPU foi sete vezes mais lenta que a CPU, mas com um maior número de galáxias, a GPU tem um ganho significativo, com cem mil galáxias, o GPU é 116 mais rápida que a CPU nos cálculos.[12]

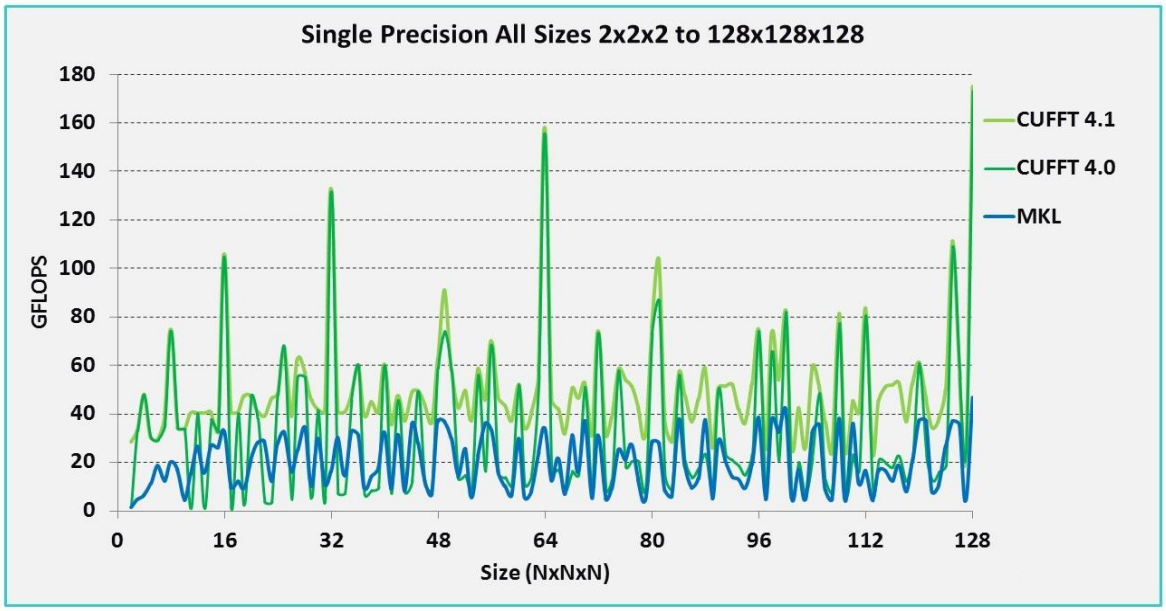

FIGURA 5 – Transformada de Fourier, comparação entre a cuFFT 4.1 em Tesla M2090, ECC on e MKL 10.2.3, TYAN FT72-B7015 Xeon x5680 Six-Core@ 33.3Ghz

Fonte: NVIDIA [13]

2.3.1 *Bibliotecas CUDA*

2.3.2 *cuFFt*

A NVIDIA CUDA Fast Fourier Transform library(cuFFT) fornece uma interface para calcular FFT até 10 vezes mais rápido que o MKL Figura 5. Usando centenas de núcleos de processamento dentro nas GPUs NVIDIA, a cuFFT oferece o desempenho de ponto flutuante de uma GPU sem a necessidade de desenvolver a sua própria implementação em CUDA.[13]

Largamente utilizada em aplicações que vão desde a física computacional, processamentos de imagens e processamento de sinais, a transformada rápida de Fourier, é um algoritmo eficiente para calcular as transformadas de Fourier discretas de um conjunto de dados de valor complexo ou real. A cuFFT usa algoritmos baseados no algoritmos de Cooley-Tukey e Bluestein.[13]

2.3.3 cuBLAS

A biblioteca da NVIDIA CUDA para Basic Linear Algebra Subroutines, a cuBLAS, é uma versão acelerada para GPU da biblioteca padrão BLAS e proporciona um desempenho de 6 a 17 vezes melhor que a mais recente MKL BLAS Figura 6 na página seguinte. No novo CUDA 6.0, a biblioteca cuBLAS-XT, tem suporte a múltiplas GPUs.[14]

Os principais recursos da cuBLAS são:

- Suporte para todas as 152 rotinas BLAS.
- Suporta variáveis single, double, e double complex.
- Suporte a Fortran.
- Suporte a múltiplas GPUs para o mesmo Kernel.
- Podem chamar funções no próprio Kernel
- API para processamento em lote da fatoração LU
- Processamento em lote da GEMM API
- Nova implementação da TRSV (Optimizing triangular solve), 7 vezes mais rápida que a anterior.

2.3.4 cuBLAS-XT

A cuBLAS-XT é um conjunto de rotinas que aceleram a BLAS (Basic Linear Algebra Subroutine) dividindo o trabalho em mais de uma GPU. Usando um design de streaming, a cuBLAS-XT gerencia com eficiência transferências em todo o barramento PCI-Express automaticamente, o que permite que os dados de entrada e saída sejam armazenados na memória do host. Isto proporciona uma operação out-of-core, em que o limite de memória passa a ser do Host e não mais o da GPU.

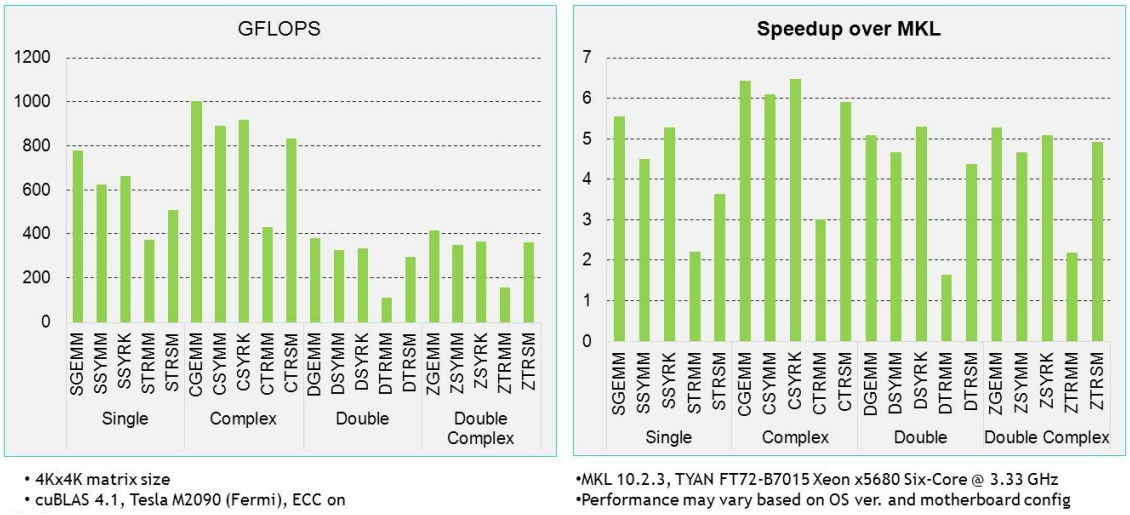

FIGURA 6 – Velocidade de processamento da cuBLAS e quantidade de vezes que é mais rápida que a MKL BLAS

Fonte: NVIDIA [14]

Vindo a partir do CUDA 6.0, uma versão gratuita do cuBLAS-XT está incluída no kit de ferramentas CUDA, como parte da biblioteca cuBLAS. A versão gratuita suporta o processamento em GPUs individuais e placas dual-GPU, como a Tesla K10 ou GeForce GTX690.

A versão principal do cuBLAS-XT suporta divisão do trabalho em vários GPUs conectados à mesma placa-mãe, com quase perfeito escalonamento quanto mais GPUs são adicionados. Um único sistema com 4 GPUs Tesla K40 é capaz de atingir mais de 4,5 TFLOPS de desempenho em precisão dupla.[15]

2.3.5 *cuSPARSE*

A biblioteca NVIDIA CUDA Sparse Matrix library - cuSPARSE fornece um conjunto de subrotinas de álgebra linear básica para matrizes esparsas que proporciona um desempenho de até 8 vezes mais rápido que o mais recente MKL. A cuSPARSE foi desenvolvida para se chamada a partir de códigos em C / C++ e a ultima versão inclui rotinas para solucionar matrizes triangulares esparsas.[16]

Os principais recursos da cuSPARSE são:

- Suporte matrizes nos formatos: COO, CSR, CSC, ELL/HYB and Blocked

CSR.

- Nível 1 rotinas para operações entre vetor esparso x vetor cheio
- Nível 2 rotinas para operações entre vetor esparso x vetor cheio
- Nível 3 rotinas para operações entre vetor esparso x multiplos de vetores cheio
- Rotinas para multiplicação e adição de matrizes esparsas
- Rotinas que permite a conversão entre matrizes de diferentes formatos
- Rotinas para matriz esparsa triangular
- Rotinas para solução de matriz tridiagonal

2.3.6 *cuRAND*

A NVIDIA CUDA Random Number Generation library - cuRAND proporciona alto desempenho na geração de números aleatórios. A cuRAND oferece números aleatórios de alta qualidade 8 vezes mais rápido usando as centenas de processadores disponíveis nas GPUs NVIDIA.

A cuRAND fornece duas interfaces flexíveis, permitido que se gere números aleatórios chamando a partir do código no host, ou a parti de dentro das funções CUDA no Kernel rodado na GPU. A variedade de algoritmos RNG e opções de distribuição para que possa ser escolhida a melhor solução para cada necessidade.

2.3.7 *CULA*

A CULA é uma biblioteca de álgebra linear para as GPUs de arquitetura NVIDIA CUDA, desenvolvida para melhorar e acelerar os cálculos matemáticos mais sofisticados. A CULA não exige experiência de programação em CUDA, o que a torna fácil de usar.

A CULA é composta de três versões: CULA Basic, CULA Premium e CULA Commercial. Em cada versão são oferecidos diferentes níveis de funcionalidades,

recursos e suporte. Não há custo para o uso da versão básica e o usuários podem redistribuí-la sem restrições. As versões CULA Premium e CULA Commercial oferecem rotinas adicionais e permitir as empresas que usem as rotinas internamente ou para criação de produtos.[17]

2.3.8 MAGMA

O projeto MAGMA pretende desenvolver uma biblioteca de álgebra linear similar ao LAPACK para arquiteturas heterogêneas/híbridas, começando pelo sistema "Multicore + GPU". O desenvolvimento da MAGMA é baseado na idéia de que para enfrentar os complexos desafios dos ambientes híbridos, as soluções de software ideais deverão ser também híbridas, combinando os pontos fortes de diferentes algoritmos num único modelo. Com base nessa idéia, o objetivo é projetar algoritmos de álgebra linear e estruturas para sistemas manycore e GPU híbridos que podem permitir que os aplicativos explorem plenamente o poder que cada um dos componentes do sistema.[18]

3 Computação Paralela de Autovalores

3.1 Autovetores E Autovalores

Os conceitos de Autovalores e Autovetores, também chamados de valores próprios e vetores próprios ou ainda, valores característicos e vetores característicos são utilizados em muitas áreas[19]. Os autovalores e autovetores são amplamente utilizados em álgebra matricial, quando deseja-se observar determinadas características de uma matriz. Pode-ser observar através dos cálculos de autovalores e autovetores as freqüências naturais e modos de vibração de varias estruturas, como pontes, por exemplo.

Definição: Se A é uma matriz quadrada nxn, um escalar λ será um autovalor de A se satisfazer:

$$A\mathbf{u} = \lambda\mathbf{u} \tag{3.1}$$

Se $\mathbf{u}$ é um vetor que satisfaz a condição 3.1 e $\mathbf{u} \neq 0$, então $\mathbf{u}$ é um autovetor de A. O conjunto de todos os valores de λ de A são autovalores de A e para cada λ_i teremos um vetor correspondente $\mathbf{u_i}$.

Os autovalores λ são as raízes do polinômio característico:

$$det(A - \lambda I)\mathbf{u} = 0 \tag{3.2}$$

Os autovalores podem ser números reais ou complexos, para uma matriz simétrica, garante-se que os autovalores são reais.

Os autovalores de uma matriz quadrada A são as raízes da correspondente

equação característica. A matriz A tem pelo menos um autovalor e no máximo n autovalores numericamente diferentes. Os autovalores devem ser determinados primeiro. Com os autovalores determinados, os correspondentes autovetores são obtidos resolvendo o sistema de equações lineares.

3.2 Método de Diferenças Finitas

3.2.1 Aproximações por Diferenças Finitas

A solução de uma equação diferencial (ED)implica na determinação dos valores da variável dependente em cada ponto do intervalo de interesse. Computacionalmente, podemos lidar apenas com uma região contínua se for possível determinar uma expressão analítica para a solução da equação diferencial. Nesse caso, o computador pode ser utilizado para calcular a solução em qualquer ponto desejado da região, com o uso da solução analítica. Contudo, no caso de técnicas numéricas de solução, não é possível tratar a região de interesse como contínua, já que, em geral, os métodos numéricos obtêm a solução do problema em pontos preestabelecidos. O processo de transformação do domínio contínuo em domínio discreto é chamado de *discretização*, onde o conjunto de pontos discretos escolhidos para representar o domínio de interesse é chamado de *malha*. A Figura 7 mostra uma malha de pontos uniformemente espaçados, o passo da malha é definido como sendo a distância entre dois pontos adjacentes e é dado por $\Delta x = x_i - x_{i-1}$.passo da malha em x

Para que seja possível tratar numericamente as equações diferenciais, elas devem ser expressas sob a forma de operações aritméticas que o computador possa executar. Essencialmente, devemos representar os operadores diferenciais (contínuos) presentes na ED por expressões algébricas (discretas), ou seja,

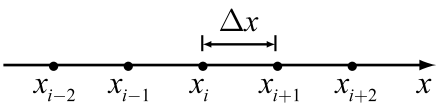

Figura 7 – Malha de pontos uniformemente espaçados.

devemos discretizar a ED . Portanto, antes de resolver a ED numericamente é preciso encontrar, para os termos que nela aparecem, as respectivas expressões escritas em função dos pontos (finitos) da malha. Essas expressões são denominadas *aproximações por diferenças finitas*. O resultado final desse processo é uma equação algébrica denominada EDF . A EDF é escrita para cada ponto da região discretizada em que se deseja calcular a solução do problema. Resolvendo-se as EDFs, encontra-se a solução aproximada do problema. Tal solução não é exata devido a (*i*) erros inerentes ao processo de discretização das equação, (*ii*) erros de arredondamento nos cálculos feitos pelo computador e (*iii*) erros na aproximação numérica das condições auxiliares.

Pode-se obter uma aproximação de diferenças finitas diretamente da definição de derivada de uma função f contínua,

$$\frac{\mathrm{d}f}{\mathrm{d}x} = \lim_{\Delta x \to 0} \frac{f(x+\Delta x) - f(x)}{\Delta x}. \tag{3.3}$$

Para tanto, basta que Δx assuma um valor fixo (não-nulo), ao invés de tender a zero, para que o lado direito da Eq. (3.3) represente uma aproximação (avançada) de diferenças finitas:

$$\frac{\mathrm{d}f}{\mathrm{d}x} \approx \frac{f(x+\Delta x) - f(x)}{\Delta x}. \tag{3.4}$$

Desse modo, utilizando-se dois valores de f separados por uma distância (finita) Δx, a expressão (3.4) representa uma aproximação algébrica para a primeira derivada de f. Essa situação está ilustrada na Figura 8 na página seguinte, onde os dois pontos, x e $x + \Delta x$, afastados entre si por uma distância Δx, formam a reta secante cuja declividade serve de aproximação para a derivada da função f no ponto x. Quando a separação Δx diminui, a reta secante se aproxima da reta tangente (derivada real), melhorando assim o valor estimado para a derivada.

Aproximações de diferenças finitas, como a mostrada anteriormente, podem ser obtidas de várias formas. As técnicas mais comuns são a expansão em série de Taylor e a interpolação polinomial. O método da expansão em série de Taylor será utilizado na obtenção de aproximações de diferenças finitas de primeira e segunda ordem para as derivadas primeira, segunda e mista de uma função f. A técnica de interpolação é geralmente utilizada no contexto em que se faz

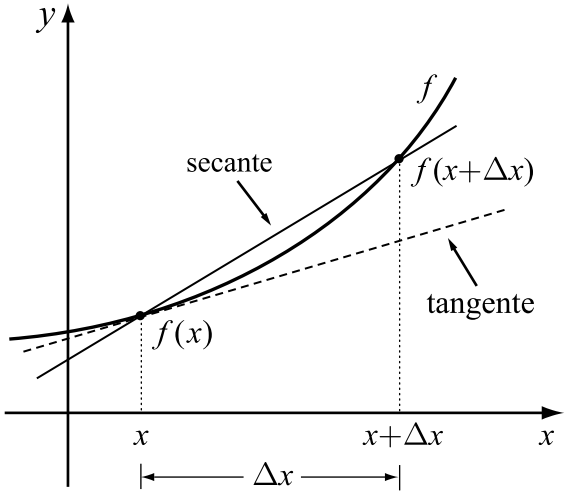

FIGURA 8 – Aproximação da derivada primeira da função genérica f no ponto x por uma reta secante.

necessário o uso de malhas cujo espaçamento não é uniforme.

3.2.2 *Aproximações para a Derivada Primeira*

As aproximações de diferenças finitas têm como base a expansão em série de Taylor de uma função f. Supondo que f seja contínua no intervalo $[a, b]$ de interesse e que possua derivadas até ordem N contínuas nesse intervalo, o teorema de Taylor nos permite escrever, para todo ponto $x \in [a, b]$,

$$f(x) = f(x_0) + (\Delta x)\left.\frac{\mathrm{d}f}{\mathrm{d}x}\right|_{x_0} + \frac{(\Delta x)^2}{2!}\left.\frac{\mathrm{d}^2 f}{\mathrm{d}x^2}\right|_{x_0} + \frac{(\Delta x)^3}{3!}\left.\frac{\mathrm{d}^3 f}{\mathrm{d}x^3}\right|_{x_0} + \cdots + R_N, \quad (3.5)$$

em que $\Delta x = x - x_0$ e R_N é o resto (de Lagrange), definido como

$$R_N = \frac{(\Delta x)^N}{N!}\left.\frac{\mathrm{d}^N f}{\mathrm{d}x^N}\right|_{\xi}, \quad \xi \in [a, b]. \quad (3.6)$$

Para aproximar a derivada primeira de uma função f no ponto x_i vamos expandir $f(x_i + \Delta x)$ em série de Taylor em torno do ponto x_i,

$$f(x_i + \Delta x) = f(x_i) + (\Delta x)\left.\frac{\mathrm{d}f}{\mathrm{d}x}\right|_{x_i} + \frac{(\Delta x)^2}{2!}\left.\frac{\mathrm{d}^2 f}{\mathrm{d}x^2}\right|_{x_i} + \frac{(\Delta x)^3}{3!}\left.\frac{\mathrm{d}^3 f}{\mathrm{d}x^3}\right|_{x_i} + \cdots, \quad (3.7)$$

onde as reticências indicam os termos restantes da série de Taylor até o resto R_N.

Após isolar a primeira derivada, podemos escrever

$$\left.\frac{\mathrm{d}f}{\mathrm{d}x}\right|_{x_i} = \frac{f(x_i+\Delta x)-f(x_i)}{\Delta x} + \left[-\frac{(\Delta x)}{2!}\left.\frac{\mathrm{d}^2 f}{\mathrm{d}x^2}\right|_{x_i} - \frac{(\Delta x)^2}{3!}\left.\frac{\mathrm{d}^3 f}{\mathrm{d}x^3}\right|_{x_i} - \cdots\right]. \quad (3.8)$$

Note que, para isolar a primeira derivada, todos os termos da série de Taylor foram divididos pelo espaçamento Δx. Podemos então dizer que a primeira derivada é igual ao quociente

$$\frac{f(x_i+\Delta x)-f(x_i)}{\Delta x}, \quad (3.9)$$

mais o erro local de truncamento (ELT), dado por:

$$\left[-\frac{(\Delta x)}{2!}\left.\frac{\mathrm{d}^2 f}{\mathrm{d}x^2}\right|_{x_i} - \frac{(\Delta x)^2}{3!}\left.\frac{\mathrm{d}^3 f}{\mathrm{d}x^3}\right|_{x_i} - \cdots\right]. \quad (3.10)$$

O ELT aparece naturalmente devido à utilização de um número finito de termos na série de Taylor. Como não podemos tratar os infinitos termos dessa série na aproximação numérica para a derivada de f, a série foi truncada a partir da derivada de segunda ordem inclusive. O ELT fornece uma medida da diferença entre o valor exato da derivada e sua aproximação numérica, indicando também que essa diferença varia linearmente com a redução do espaçamento Δx, isto é, com o refinamento da malha. Assim, para reduzirmos o erro por quatro, por exemplo, devemos utilizar um espaçamento $1/4$ do original e portanto, quatro vezes mais pontos na malha. Dessa forma, os termos do ELT serão representados por $O(\Delta x)$. Deve-se notar que uma expressão do tipo $O(\Delta x)$ só indica como ELT varia com o refinamento da malha, e não o valor do erro.

Podemos simplificar a notação se escrevendo f_i para $f(x_i)$ ou, em geral, $f_{i\pm k}$ para $f(x_i \pm k\Delta x)$. Com isso a Eq. (3.8) se torna

$$\left.\frac{\mathrm{d}f}{\mathrm{d}x}\right|_{x_i} = \frac{f_{i+1}-f_i}{\Delta x} + O(\Delta x), \quad (3.11)$$

que é uma equação de diferenças finitas que representa uma aproximação de primeira ordem para a primeira derivada de f, utilizando diferença avançada, visto que no cálculo da derivada no ponto x_i foi utilizado um ponto adiante de

x_i, no caso, x_{i+1}. A declividade (primeira derivada) de f em x_i é aproximada pela declividade da reta secante formada pelos pontos (x_i, f_i) e (x_{i+1}, f_{i+1}), conforme mostra a Figura 9.

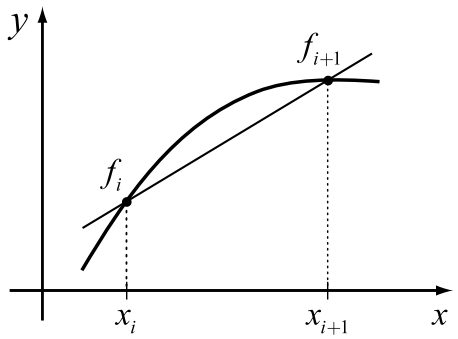

FIGURA 9 – Pontos utilizados na aproximação para a primeira derivada de f por diferença avançada.

Uma segunda aproximação de diferenças finitas pode ser obtida a partir da expansão de $f(x_i - \Delta x)$ em série de Taylor em torno do ponto x_i:

$$f(x_i - \Delta x) = f(x_i) - (\Delta x) \left.\frac{\mathrm{d}f}{\mathrm{d}x}\right|_{x_i} + \frac{(\Delta x)^2}{2!} \left.\frac{\mathrm{d}^2 f}{\mathrm{d}x^2}\right|_{x_i} + O(\Delta x)^3. \tag{3.12}$$

Isolando a primeira derivada, temos

$$\left.\frac{\mathrm{d}f}{\mathrm{d}x}\right|_{x_i} = \frac{f_i - f_{i-1}}{\Delta x} + O(\Delta x), \tag{3.13}$$

que é outra aproximação de primeira ordem para a primeira derivada de f. Diferentemente da Eq. (3.11), na qual utiliza-se um ponto adiante de x_i, a Eq. (3.13) utiliza o ponto x_{i-1}, ponto este anterior a x_i. Por essa razão, essa equação de diferenças é chamada de aproximação por diferenças atrasadas. A Figura 10 na página seguinte mostra os pontos utilizados nessa aproximação. A declividade da função f no ponto x_i é aproximada pela declividade da reta secante aos pontos (x_{i-1}, f_{i-1}) e (x_i, f_i).

Até agora obtivemos somente aproximações de primeira ordem para a derivada primeira de f, uma aproximação de $O(\Delta x)^2$ ainda para a primeira derivada pode ser obtida ao subtrairmos as expansões em série de Taylor representadas

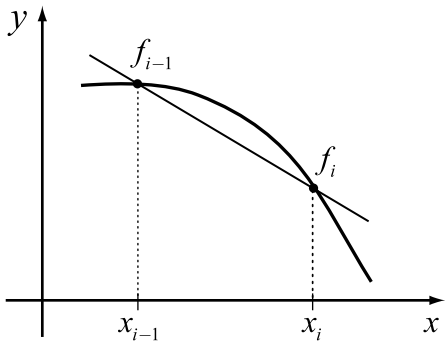

FIGURA 10 – Pontos utilizados na aproximação para a primeira derivada de f por diferença atrasada.

pelas Eqs. (3.7) and (3.12):

$$f(x_i + \Delta x) - f(x_i - \Delta x) = 2(\Delta x) \left.\frac{\mathrm{d}f}{\mathrm{d}x}\right|_{x_i} + O(\Delta x)^3, \tag{3.14}$$

ou, isolando a derivada,

$$\left.\frac{\mathrm{d}f}{\mathrm{d}x}\right|_{x_i} = \frac{f_{i+1} - f_{i-1}}{2\Delta x} + O(\Delta x)^2. \tag{3.15}$$

Note que a aproximação dada pela Eq. (3.15) utiliza os pontos x_{i-1} e x_{i+1} para o cálculo da primeira derivada de f no ponto central x_i. Por essa razão, ela é denominada aproximação por diferenças centrais. Neste caso, conforme mostra a Figura 11 na próxima página, a derivada de f em x_i é aproximada pela declividade da reta secante que passa pelos pontos (x_{i-1}, f_{i-1}) e (x_{i+1}, f_{i+1}).

No caso de aproximações de segunda ordem, reduções sucessivas no passo Δx da malha provocam uma redução quadrática no erro da aproximação da primeira derivada de f pela Eq. (3.15). Ao dividirmos o passo por dois, por exemplo, o erro é dividido por quatro, sem precisarmos de quatro vezes mais pontos, como nas expressões de primeira ordem. Isso é uma propriedade extremamente útil, já que, com menor número de pontos e, portanto, menor esforço computacional, podemos conseguir uma aproximação melhor que aquelas fornecidas pelas Eq. (3.11) e Eq. (3.13).

Em resumo, as três fórmulas para a primeira derivada de f deduzidas anteri-

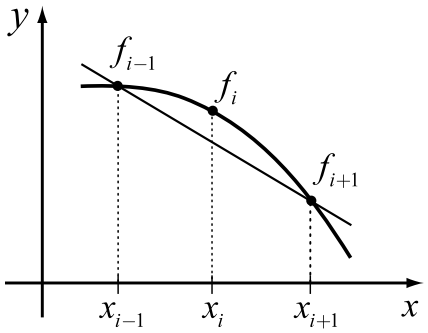

FIGURA 11 – Pontos utilizados na aproximação de segunda ordem para a primeira derivada de f por diferença central.

ormente, a partir da expansão em série de Taylor, são:

$$\left.\frac{\mathrm{d}f}{\mathrm{d}x}\right|_{x_i} \approx \frac{f_{i+1}-f_i}{\Delta x}, \qquad \text{(fórmula avançada)}$$

$$\left.\frac{\mathrm{d}f}{\mathrm{d}x}\right|_{x_i} \approx \frac{f_i-f_{i-1}}{\Delta x} \qquad \text{(fórmula atrasada)}$$

e

$$\left.\frac{\mathrm{d}f}{\mathrm{d}x}\right|_{x_i} \approx \frac{f_{i+1}-f_{i-1}}{2\Delta x}. \qquad \text{(fórmula central)}$$

3.2.3 *Aproximações Para A Derivada Segunda*

Expressões para derivadas de ordem superior podem ser obtidas com o mesmo procedimento com o qual obtivemos as fórmulas para a derivada primeira, isto é, por meio de manipulações adequadas da série de Taylor. Como exemplo, vamos determinar uma aproximação de diferenças centrais de segunda ordem para a segunda derivada de f. Para tal vamos utilizar ainda as Eqs. (3.7) and (3.12). Queremos combiná-las para que a primeira derivada de f seja eliminada, pois estamos interessados na segunda derivada. Por sua vez, as derivadas de ordem superior a dois que permanecerem na expansão farão parte do ELT. Assim,

$$f(x_i+\Delta x)-f(x_i-\Delta x) = 2f(x_i)+(\Delta x)^2\left.\frac{\mathrm{d}^2f}{\mathrm{d}x^2}\right|_{x_i}+O(\Delta x)^4. \qquad (3.16)$$

Rearranjando os termos, obtemos

$$\left.\frac{\mathrm{d}^2 f}{\mathrm{d}x^2}\right|_{x_i} = \frac{f_{i+1} - 2f_i + f_{i-1}}{(\Delta x)^2} + O(\Delta x)^2, \tag{3.17}$$

que é a fórmula de diferenças finitas centrais de segunda ordem para derivadas segundas. A Eq. (3.17) é a aproximação mais comum encontrada na literatura para derivadas de segunda ordem.

Aproximações de diferenças avançadas e atrasadas de $O(\Delta x)$ para a segunda derivada podem ser obtidas manipulando-se convenientemente as expansões de $f(x_i \pm \Delta x)$ e $f(x_i \pm 2\Delta x)$, nesse caso toma-se os sinais positivos para a aproximação avançada e os negativos para a atrasada. Aproximações avançadas e atrasadas de ordem superior podem também ser obtidas pelas expansões em série de Taylor, simplesmente utilizando mais termos dessa série.

Nos casos em que o problema é dependente do tempo devemos considerar a expansão em série de Taylor de uma função de duas variáveis independentes, supondo que $f = f(x,t)$, a expansão em torno do ponto x_i fornece:

$$f(x_i + \Delta x, t) = f(x_i, t) + (\Delta x) \left.\frac{\partial f}{\partial x}\right|_{x_i} + \frac{(\Delta x)^2}{2!} \left.\frac{\partial^2 f}{\partial x^2}\right|_{x_i} + \cdots, \tag{3.18}$$

da mesma forma,

$$f(x_i - \Delta x, t) = f(x_i, t) - (\Delta x) \left.\frac{\partial f}{\partial x}\right|_{x_i} + \frac{(\Delta x)^2}{2!} \left.\frac{\partial^2 f}{\partial x^2}\right|_{x_i} + \cdots. \tag{3.19}$$

Assim, podemos gerar aproximações de primeira ou segunda ordem para as derivadas parciais, como por exemplo:

$$\left.\frac{\partial f}{\partial x}\right|_{x_i} = \frac{f(x_i + \Delta x, t) - f(x_i, t)}{\Delta x} + O(\Delta x), \qquad \text{(avançada)} \tag{3.20}$$

$$\left.\frac{\partial f}{\partial x}\right|_{x_i} = \frac{f(x_i, t) - f(x_i - \Delta x, t)}{\Delta x} + O(\Delta x), \qquad \text{(atrasada)} \tag{3.21}$$

$$\left.\frac{\partial f}{\partial x}\right|_{x_i} = \frac{f(x_i + \Delta x, t) - f(x_i - \Delta x, t)}{2\Delta x} + O(\Delta x)^2 \qquad \text{(central)} \tag{3.22}$$

e

$$\left.\frac{\partial^2 f}{\partial x^2}\right|_{x_i} = \frac{f(x_i+\Delta x,t) - 2f(x_i,t) + f(x_i-\Delta x,t)}{\Delta x^2} + O(\Delta x)^2. \quad \text{(central)} \qquad (3.23)$$

É fácil mostrar que existem expressões equivalentes para o tempo, ou seja, Δt intervalo de tempo ou passo da malha em t

$$\left.\frac{\partial f}{\partial t}\right|_{t_i} = \frac{f(x,t_i+\Delta t) - f(x,t_i)}{\Delta t} + O(\Delta t), \quad \text{(avançada)} \qquad (3.24)$$

$$\left.\frac{\partial f}{\partial t}\right|_{t_i} = \frac{f(x,t_i) - f(x,t_i-\Delta t)}{\Delta t} + O(\Delta t), \quad \text{(atrasada)} \qquad (3.25)$$

$$\left.\frac{\partial f}{\partial t}\right|_{t_i} = \frac{f(x,t_i+\Delta t) - f(x,t_i-\Delta t)}{2\Delta t} + O(\Delta t)^2 \quad \text{(central)} \qquad (3.26)$$

e

$$\left.\frac{\partial^2 f}{\partial t^2}\right|_{t_i} = \frac{f(x,t_i+\Delta t) - 2f(x,t_i) + f(x,t_i-\Delta t)}{\Delta t^2} + O(\Delta t)^2. \quad \text{(central)} \qquad (3.27)$$

Com as expansões em série de Taylor de $f(x,y)$ em torno do ponto (x_i, y_j)— em que o índice j está associado à coordenada y—, podemos também determinar expressões envolvendo derivadas parciais mistas com outras variáveis espaciais, isto é, do tipo $\frac{\partial^2 f}{\partial x \partial y}$.

Como estamos interessados agora em obter uma expressão que relacione a variação de f com incrementos em x e y, simultaneamente, devemos utilizar a expansão em série de Taylor de funções de duas variáveis, dada por:

$$\begin{aligned} f(x_i+\Delta x, y_j+\Delta y) = f(x_i,y_j) + (\Delta x)\left.\frac{\partial f}{\partial x}\right|_{x_i,y_j} + (\Delta y)\left.\frac{\partial f}{\partial y}\right|_{x_i,y_j} + \frac{(\Delta x)^2}{2!}\left.\frac{\partial^2 f}{\partial x^2}\right|_{x_i,y_j} \\ + 2\frac{(\Delta x)(\Delta y)}{2!}\left.\frac{\partial^2 f}{\partial x \partial y}\right|_{x_i,y_j} + \frac{(\Delta y)^2}{2!}\left.\frac{\partial^2 f}{\partial y^2}\right|_{x_i,y_j} + \cdots . \end{aligned} \qquad (3.28)$$

Após algumas manipulações algébricas, a combinação adequada das expansões de $f(x_i \pm \Delta x, y_j - \Delta y)$ e $f(x_i \pm \Delta x, y_j + \Delta y)$ até termos de segunda ordem fornece

$$\left.\frac{\partial^2 f}{\partial x \partial y}\right|_{x_i,y_j} = \frac{f_{i+1,j+1} - f_{i+1,j-1} - f_{i-1,j+1} + f_{i-1,j-1}}{4(\Delta x)(\Delta y)} + O[(\Delta x)^2(\Delta y)^2]. \qquad (3.29)$$

3.3 Autovalores de um Poço Quantico Simétrico

A equação de Schrödinger unidimensional independente do tempo para uma partícula de massa m^* se movendo num potencial $V(x)$, é dada por:

$$-\frac{\hbar^2}{2m^*}\frac{\mathrm{d}^2\psi(x)}{\mathrm{d}x^2} + V(x)\psi(x) = E\psi(x), \tag{3.30}$$

onde $\hbar = h/2\pi$ é a constante reduzida de Plank. Como sabemos esta é uma equação de autovalor, na qual ψ representa as autofunções e E os autovalores de energia. Vamos impor as seguintes condições de contorno à Eq. (3.30):

$$\psi(-\infty) = 0 \quad \text{e} \quad \psi(\infty) = 0. \tag{3.31}$$

O primeiro passo na resolução do problema é impor uma malha no domínio de solução e obter a equação de diferenças finitas apropriada. Iremos então procurar a solução da equação de diferenças nos pontos da malha. Visto que estamos substituindo uma equação diferencial contínua por uma equação de diferenças finitas discreta, devemos procurar a solução em uma malha finita. Esperamos, é claro, que a solução desse problema seja (aproximadamente) a solução do problema original. Nesse exemplo iremos tomar a malha associada a um certo espaçamento Δx e obter soluções para os pontos interiores as soluções para $x = -\infty$ e $x = \infty$ estão fixadas pelas condições de contorno e, portanto, não estão sujeitas à alterações. Substituindo os operadores diferenciais na equação diferencial original, Eq. (3.30), pelas aproximações dadas pelas Eqs. (3.15) and (3.17), obtemos a seguinte equação de diferenças finitas:

$$-\frac{\hbar^2}{2m^*}\left[\frac{\psi_{i+1} - 2\psi_i + \psi_{i-1}}{(\Delta x)^2}\right] + V_i\psi_i = E_m\psi_i, \quad i = 1, 2, \ldots, n-1, \tag{3.32}$$

onde $\Delta x = x_i - x_{i-1}$ é o passo da malha e é quantidade de intervalos da malha ($n+1$ pontos). As Eqs. (3.32) formam um sistema linear com $n-1$ equações a $n-1$ incógnitas $\psi_1, \ldots, \psi_{n-1}$. Tal sistema pode ainda ser escrito na forma

$$-\frac{\hbar^2}{2m^*}\psi_{i-1} + \left[\frac{\hbar^2}{m^*} + (\Delta x)^2 V_i\right]\psi_i - \frac{\hbar^2}{2m^*}\psi_{i+1} = (\Delta x)^2 E_j\psi_i, \tag{3.33}$$

que, por sua vez, pode ser escrito na forma matricial $\mathbf{A}\psi = \lambda\psi$, ou seja:

$$\begin{bmatrix} d_1 & e_1 & 0 & \cdots & 0 \\ e_1 & d_2 & e_2 & \cdots & 0 \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ 0 & \cdots & e_{n-1} & d_{n-2} & e_{n-2} \\ 0 & \cdots & 0 & e_{n-2} & d_{n-1} \end{bmatrix} \begin{bmatrix} \psi_1 \\ \psi_2 \\ \vdots \\ \psi_{n-2} \\ \psi_{n-1} \end{bmatrix} = (\Delta x)^2 E_j \begin{bmatrix} \psi_1 \\ \psi_2 \\ \vdots \\ \psi_{n-2} \\ \psi_{n-1} \end{bmatrix},$$

onde os elementos da diagonal principal e os elementos não-nulos fora da diagonal principal são dados, respectivamente, por

$$d_i = \frac{\hbar^2}{m^*} + (\Delta x)^2 V_i$$

e

$$e_i = -\frac{\hbar^2}{2m^*} = \text{constante},$$

em que seus autovalores são dados por $\lambda = (\Delta x)^2 E_i$. A matriz obtida com esse exemplo é dita esparsa por possuir muitos elementos nulos e também pode ser classificada ainda como real, tridiagonal e simétrica.

Para obter os autovalores e autovetores da matriz $\mathbf{A}$ devemos resolver o sistema de equações lineares representado pela Eq. (3.33). Existem duas classes de métodos numéricos que podem ser utilizados para este fim, numa classe encontram-se os métodos diretos, na outra os indiretos ou iterativos. Com os métodos diretos, a solução é computada pela execução de um número finito de operações aritméticas, muitas vezes esse número de operações é conhecido *a priori*. Já com os métodos iterativos, partindo de uma tentativa inicial de solução, uma seqüência de aproximações é gerada com o propósito de convergir para a solução do problema[4].

3.4 Matriz Tridiagonal

A resolução da equação de Schrödinger independente do tempo usando o método das diferença finitas gera uma matriz real, tridiagonal e simétrica conforme demonstrado em [4], no apêndice B.

Uma matriz quadrada **Q** é definida como tridiagonal simétrica se seus elementos $\mathbf{Q_{ij}}$ são iguais a 0 sempre que |i-j|>1 e quanto tem os elementos da diagonal superior iguais aos do diagonal inferior.

$$\begin{bmatrix} d_n & e_n & 0 & \cdots & 0 \\ e_n & d_{n+1} & e_{n+1} & \cdots & 0 \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ 0 & \cdots & e_{n-1} & d_{n-2} & e_{n-2} \\ 0 & \cdots & 0 & e_{n-2} & d_{n-1} \end{bmatrix}$$

3.5 Teorema dos Discos de Gershgorin

Se A é uma matriz n x n, A = D + F, com D tendo os elementos de sua diagonal iguais ao da diagonal de A e todos os outros elementos nulos e F tendo a diagonal nula e todos os elementos restantes iguais aos de A, tal que:

$$\begin{bmatrix} d_n & e_n & 0 & \cdots & 0 \\ e_n & d_{n+1} & e_{n+1} & \cdots & 0 \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ 0 & \cdots & e_{n-1} & d_{n-2} & e_{n-2} \\ 0 & \cdots & 0 & e_{n-2} & d_{n-1} \end{bmatrix} = \begin{bmatrix} d_n & 0 & 0 & \cdots & 0 \\ 0 & d_{n+1} & 0 & \cdots & 0 \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ 0 & \cdots & 0 & d_{n-2} & 0 \\ 0 & \cdots & 0 & 0 & d_{n-1} \end{bmatrix} + \begin{bmatrix} 0 & e_n & 0 & \cdots & 0 \\ e_n & 0 & e_{n+1} & \cdots & 0 \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ 0 & \cdots & e_{n-1} & 0 & e_{n-2} \\ 0 & \cdots & 0 & e_{n-2} & 0 \end{bmatrix}$$

Se na matriz F os valores dos termos são muito pequenos em relação aos elementos da matriz D, então os autovalores não devem se distanciar muito dos elementos da matriz D. O termo muito pequeno, deve ser entendido como relativamente aos autovalores, ou melhor, aos elementos da matriz D[20].

Se em (3.1) considerarmos um autovalor fixo e um autovetor correspondente, temos que:

$$\sum_{j=1}^{N} d_{ij} \quad x^j \quad = \quad i = 1, \cdots, N. \tag{3.34}$$

Como pretendemos avaliar a distância de λ a diagonal de A, é natural reescrever da seguinte forma:

$$\lambda x^i - d_{ii}x^i = \sum_{j=1}^{N} a_{ij} \quad x^j \tag{3.35}$$

Se x^i for nulo, nada se pode concluir sobre $\lambda - d_{ii}$, sendo não nulo, temo que:

$$\lambda x^i - d_{ii} = \sum_{j=1}^{N} d_{ij} \quad \frac{x^j}{x^i} \tag{3.36}$$

Já que escolhemos busca entre as componentes não nulas de x. Vamos agora nos afastar o máximo possivel do zero e tomar a componente que tiver o maior valor absoluto, dessa forma, $\frac{|x^j|}{|x^i|} \leq 1$ e consequentemente, temos que:

$$|\lambda x^i - d_{ii}| = \sum_{j=1}^{N} |d_{ij}| \tag{3.37}$$

Temos então que dado um dos autovalores da matriz A, achamos uma linha dessa matriz, tal que podemos garantir: a distância entre o elemento dessa linha na diagonal e o dado autovalor não ultrapassa a soma dos demais elementos da mesma linha.

A parti disso contruimos círculos que tem por centro os elementos da diagonal de A e, respectivamente, por raio, a soma dos demais elementos fora da diagonal, na correpondente linha. A união desses círculos contém todos os autovalores de A.

Sendo o raio de cada circulo dado por:

$$R_i = |\lambda x^i - d_i| \leq \sum_{j=1}^{N} |f_{ij}| \tag{3.38}$$

Para o nosso caso especifico, onde A é uma matriz tridiagonal simétrica, os autovalores de A convergem em intervalos sobre a reta real, de maneira que o i-ésimo intervalo pode ser definido como:

$$d_i - \sum_{j=1}^{N} |f_{ij}|, d_i + \sum_{j=1}^{N} |f_{ij}| \tag{3.39}$$

Então para assegurarmos que tenhamos todos os autovalores dentre os pontos escolhidos vamos usar os limites mínimo e máximo do intervalo.

$$\alpha = \frac{mim}{1 \leq i \leq n-2}(d_1 - (|e_i| + |e_{i+1}|)) \tag{3.40}$$

$$\beta = \frac{max}{1 \leq i \leq n-2}(d_1 + (|e_i| + |e_{i+1}|)) \tag{3.41}$$

Dessa forma garantimos que temos um intervalo, entre os extremos α e β e sobre a reta real com todos os autovalores da matriz A.

A demostração mais detalhada do teorema pode ser encontrada em [21] pagina 341.

3.6 Sequência de Sturm

Partiremos do seguinte problema: Como encontra o número de raízes reais de um polinômio num dado intervalo? Jacques Charles Françoes Sturm (1803-1855), matemático francês, publicou em 1829 um artigo no qual respondia esse problema, demostrara um algoritmo para resolução de mameira simples de como determinar a quantidade de raízes reais de um polinômio[22]. Este algoritmo permitia que a partir de um polinômio *f(x)* e dois números reais distintos, α e β, determinase a quatidades de raízes reais de *f(x)* no intervalo (α, β).

Seja $f(x)$ um polinômio de grau $n \geq 1$,

$$f(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_{n-1} x^{n-1} + a_n x^n,$$

onde $a_0, \cdots, a_n \in R$. A sequência de Sturm de $f(x)$ é uma sequência de polinômios

$$f_0(x), f_1(x), \cdots, f_m(x)$$

onde

$$\begin{aligned} f_0(x) &= f(x) \\ f_1(x) &= f'(x) \\ f_2(x) &= f_1(x)q_1(x) - f_0(x) \\ &\vdots \\ f_{m-1} &= f_{m-2}q_{m-2}(x) - f_{m-3}(x) \\ f_m &= f_{m-1}q_{m-1}(x) - f_{m-2}(x). \end{aligned} \tag{3.42}$$

onde $q_i(x)$, $i = 1, \cdots, m$, é o quociente da divisão de $f_{i-2}(x)$ por $f_{i-1}(x)$.

O procedimento para montagem da sequência de Sturm segundo desmostrado acima, pode ser descrito seguinte forma: faz-se $f_0(x) = f(x)$ e $f_1(x) = f'(x)$, onde $f'(x)$ é a derivada de $f(x)$. $f_2(x)$ é o resto da divisão com sinal oposto de $f_0(x)$ por $f_1(x)$. Se $f_2(x)$ for uma constante não-nula, então a sequência de Sturm está completa. Caso contrário, tem-se $f_3(x)$ igual ao resto da divisão com sinal oposto de $f_1(x)$ por $f_2(x)$. O processo deve ser realizado até o resto da divisão com sinal oposto ser uma constante não-nula. No caso chega-se ao último termo da sequência $f_m(x)$.

Seja $c \in R$ um número qualquer. Substituindo este número em todas as funções da sequência de Sturm, obtemos a sequência numérica

$$f_0(c), f_1(c), f_2(c), \cdots, f_m(c).$$

A parti desse ponto, analisamos os respectivos sinais do termos da sequência, obteremos uma sequência de sinais (desconsiderando possíveis valores nulos). O número de variações nesta sequência de sinais é chamado o número de variações na sequência de Sturm para $x = c$ e será denotado por $v(c)$.

Seja $f(x)$ um polinômio de grau $n \geq 1$, e duas constantes $\alpha, \beta \in R$ que não sejam raízes da equação $f(x) = 0$, o número de raízes(nR) distintas entre α e β, onde $\alpha < \beta$, é exatamente igual a diferença

$$nR = v(\alpha) - v(\beta) \tag{3.43}$$

onde $v(\alpha)$ e $v(\beta)$ representam o número de mudanças de sinais na sequência de valores assumidos pela sequência de Sturm, avaliada em $x = \alpha$ e $x = \beta$ e nR representa o numero de raízes no intervalo entre $x = \alpha$ e $x = \beta$.

Pensemos no seguinte polinômio: $f(x) = x^3 + 0x^2 - 3x + 1$, vamos aplicar a ela a divisões necessárias, conforme (3.42), até chegarmos a uma constante:

$$\begin{aligned} f_0(x) &= x^3 + 0x^2 - 3x + 1 \\ f_1(x) &= 3x^2 + 0x - 3 \\ f_2(x) &= 2x - 1 \\ f_2(x) &= \frac{9}{4} \end{aligned} \tag{3.44}$$

De posse da sequência gerada por esse polinômio, vamos agora calcular para

Tabela 2

x	f_0	f_1	f_2	f_3
-2	-	+	-	+
-1	+	0	-	+
0	+	-	-	+
1	-	0	+	+
2	+	+	+	+

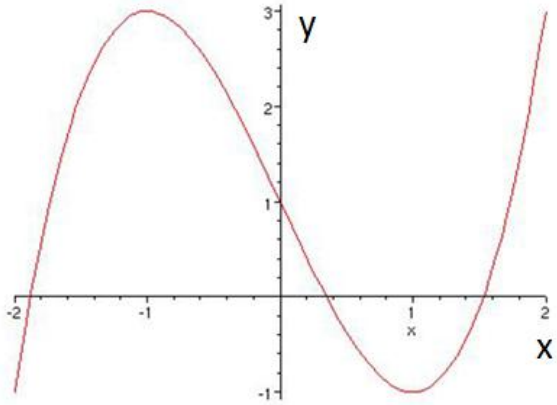

Figura 12 – Grafico do polinômio f(x)=$x^3 + x^2 + 3x + 1$

os pontos: -2, -1, 0, 1, 2 e tomar o sinal de cada um do *f(x)* para cada ponto, Tabela 2.

De posse da Tabela 2, podemos fazer uma analise de onde se encontram as raízes do polinômio através da quantidade de trocas de sinal, ao compararmos dois pontos. No ponto -2, há 3 trocas de sinais, no ponto 2, não há nenhuma, logo, entre esses dois pontos, há 3 raízes dessa equação, como pode ser visualizado graficamente em Figura 12.

Ficando claro o que é a sequência de Sturm, vamos partir agora para a sua utilidade dentro no nosso método:

Seja X, uma matriz (*nxn*), triadiagonal simétrica irredutível, e λ um número real, então a troca de sinais na sequência de Sturm, para o polinômio caracteristico dessa matriz:

$$p_0(\lambda), p_1(\lambda), ..., p_n(\lambda) \tag{3.45}$$

será igual ao número de autovalores de X menores que λ.

Temos então para essa propriedade a conversão que,$p_n(\lambda)$ tem um sinal oposto a $p_{n-1}(\lambda)$, se $p_n(\lambda) = 0$, para qualquer λ.

Uma demonstração mais detalhada pode ser encontrada em [23], na pagina 300.

Dado que a sequência de Sturm será bastante utilizada, definiremos uma função $contneg(\lambda)$, que devolvera o número de trocas de sinal para determinando (λ) na sequência de Sturm.

Givens [24] [25], foi o primeiro a dizer ser possível usar está propriedade da sequência de Sturm para calcular os autovalores de uma matiz tridiagonal simétrica.

O calculo da função $contneg(\lambda)$, associa a matriz nos permite saber quantos autovalores associados a matriz são menores que (λ). O que também nos permite subdividir o intervalo e calcular quantos autovalores há em cada intervalo fazendo: $contneg(\lambda + n)$-$contneg(\lambda)$.

Aplicando a bissecção, podemos subdividir o intervalo até que entre $contneg(\lambda)$ e $contneg(\lambda + n)$ só tenha um autovalor, e depois continua dividindo o intervalo até a precisão desejada ou um limite de interações.

Dada a matriz X, triadiagonal simétrica,

$$\begin{bmatrix} d_n & e_n & 0 & \cdots & 0 \\ e_n & d_{n+1} & e_{n+1} & \cdots & 0 \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ 0 & \cdots & e_{n-1} & d_{n-2} & e_{n-2} \\ 0 & \cdots & 0 & e_{n-2} & d_{n-1} \end{bmatrix}$$

definimos o polinômio característico de X tal que:

$$p_n(\lambda) = det(X - \lambda I) \tag{3.46}$$

Usando a definição da sequência de Sturm, montamos o polinômio característico de X em qualquer ponto do seguinte modo:

$$\begin{aligned} p_0(\lambda) &= 0 \\ p_1(\lambda) &= d_1 - \lambda \\ p_n(\lambda) &= (d_n - \lambda)p_{n-1}(\lambda) - e_{n-1}^2 p_{n-2}(\lambda) \\ n &: 2,3,...,i. \end{aligned} \tag{3.47}$$

Calculando a sequência de Sturm desta forma, quando temos um n grande, podemos ter graves problemas de *overflow/underflow*. Erro que foi detectado por [26] que propuseram a sequência de Sturm modificada para resolver este problema, que será formada pelos seguintes termos:

$$q_n(\lambda) = \frac{p_n(\lambda)}{p_{n-1}(\lambda)} \tag{3.48}$$

A parte de (3.48) se torna mais fácil calcular (3.47) usando a seguinte recorrência:

$$\begin{aligned} q_0(\lambda) &= 0 \\ q_1(\lambda) &= d_1 - \lambda \\ q_n(\lambda) &= (d_n - \lambda) - \frac{e_{n-1}^2}{q_{n-1}(\lambda)} \\ n &: 2,3,...,i. \end{aligned} \tag{3.49}$$

Desta forma resolve-se o problema de *overflow/underflow*, e passamos a contar a quantidade de sinal negativos de $q_n(\lambda)$, e não mais a troca de sinais, por isso escolhemos o nome contneg(λ) para a função.

3.7 Método da Bissecção

Este método é usado para encontrar as raízes de uma função *f(x)* contínua com raízes pertencentes aos reais definida num intervalo [a,b], tendo *f(a)* e *f(b)* sinais opostos, ou seja, *f(a)f(b)=0*. Como *f(a)* e *f(b)* têm sinais opostos e *f(x)* é contínua, pelo teorema do valor intermediário podemos afirmar que existe uma raiz neste intervalo [a,b]. O método consiste em dividir o intervalo em dois no seu ponto médio $c = \frac{(a+b)}{2}$, e então verificar em qual dos dois novos intervalos se encontra a raiz. A partir do ponto c temos 3 opções, ou *f(a).f(c)<0* e neste caso a raiz se encontra no intervalo [a,c], e tornamos c o nosso novo b, ou *f(b).f(c)<0* e neste caso

a raiz se encontra no intervalo [c,b], e tornamos *c* o nosso novo *a*, e por fim, ou *f(c)=0* e com isso sabemos que c é a raiz de nossa função.

Usando a sequência de Sturm modificada, passamos a avaliar o intervalo a ser escolhido usando a função *contNeg(x)*, nos pontos *a* e *b*, e escolhemos o intervalo da seguinte forma, se *contNeg(b)-contNeg(c)=1*, então *c* será o novo *a*, se não *c* será nosso novo *b* e repetimos o processo até a precisão requerida ou desejada.

3.7.1 Antecedentes Históricos

O método da bissecção foi originalmente proposto por Wallace Givens e originalmente chamado de método de Givens, conforme artigo publicados por Givens [24] [25]. Originalmente se implementou o método de bissecção pura a parti dos resultados da sequência de Sturm associadas ao determinante da matriz em questão. O método sequencial foi implementado e melhorado por diversos processos de aceleração pelo autores Kahan [27] e [28] e Wilkinson [29].

Wilkinson, em [23], afirma que o método da bissecção é muito bom na fase de isolamento e pode ser notadamente acelerado usando técnica de aproximação de raízes de polinômios com uma maior taxa de convergência, como o método de Newton, quadrático, ou método de Laguerre, cúbico.

Versões melhoradas do método da bissecção continuaram a aparecer, destacando-se o trabalho de Barth [26] que usa uma versão modificada da sequencia de Sturm em que se evita a maior parte dos problemas de *overflow/underflow* produzidos pela versão original. A rotina obtida foi denominada "bisect"e tem servido como base para todos o algoritmos implementados posteriormente. Sendo uma versão da rotina incluída na EISPACK [30].

Bernstein [31] aplica o método da bissecção durante a fase de isolamento e métodos mais rápidos na fase de extração. O problema de usar o método da bissecção na fase de isolamento é a possibilidade da presença de *clusters* de autovalores e nestes casos a convergência pode passa de quadrada ou cúbica a linear, e a extração dos autovalores se da em uma velocidade menor que no método da bissecção. Bernstein utiliza técnicas de interpolação para melhora a velocidade do método.

Das varias técnicas usadas durante a fase de extração, em matrizes pequenas, o melhor comportamento é oferecido pelo método de Newton [32]. Ralha, em [33] traz uma versão melhorada do método de Newton para o calculo das raízes

do polinômio característico de matrizes triagonais simétricas.

As contribuições feitas por todos os autores citados são o pano de fundo e mostram os autores que trabalharam na evolução do método da bissecção, permitindo mostrar o atual estado da arte. Como a evolução do método levou a aplicação de uma sequência de Sturm modificada, substituindo a original de Givens, resolvendo os problemas de *overflow/underflow*. O método foi claramente dividido em duas fases, a de isolamento e a de extração. Os autores citados implementaram diferentes algoritmos tentando conciliar capacidade robustez em ambas as fazes para o cálculo das raízes e tratando problemas como os *clusters* de autovalores. Definitivamente o método da bissecção é uma técnica amplamente utilizada e estudada nos casos de matrizes triagonais simétricas. Apresenta como inconveniente sua lentidão no calculo de todos os autovalores da matriz, porem oferece flexibilidade e precisão no calculo de parte do espectro[32].

Temos especial interesse no método da bissecção pelo seu alto poder de paralelização, que é o foco deste trabalho usando as placas gráficas.

3.8 Implementação

No primeiro momento, geramos as diagonais da matriz e em seguida calculamos os intervalos que contém os autovalores usando o teorema dos círculos de Gershgorin.

De posse do intervalo compreendido entre $xmin$ e $xmax$, partimos para a fase de isolamento dos intervalos que contém somente um autovalor. Em paralelo na CPU, como escolhemos os intervalos pelo seu tamanho e não pelo número de autovalores em cada, pode acontecer de algumas threads demorarem mais para termina que outras, devido ao maior número de autovalores.

De posse dos intervalos que contém somente um autovalor, partimos a fase de extração, na GPU, quantas threads forem o numero de autovalores. Um fluxograma detalhado está na

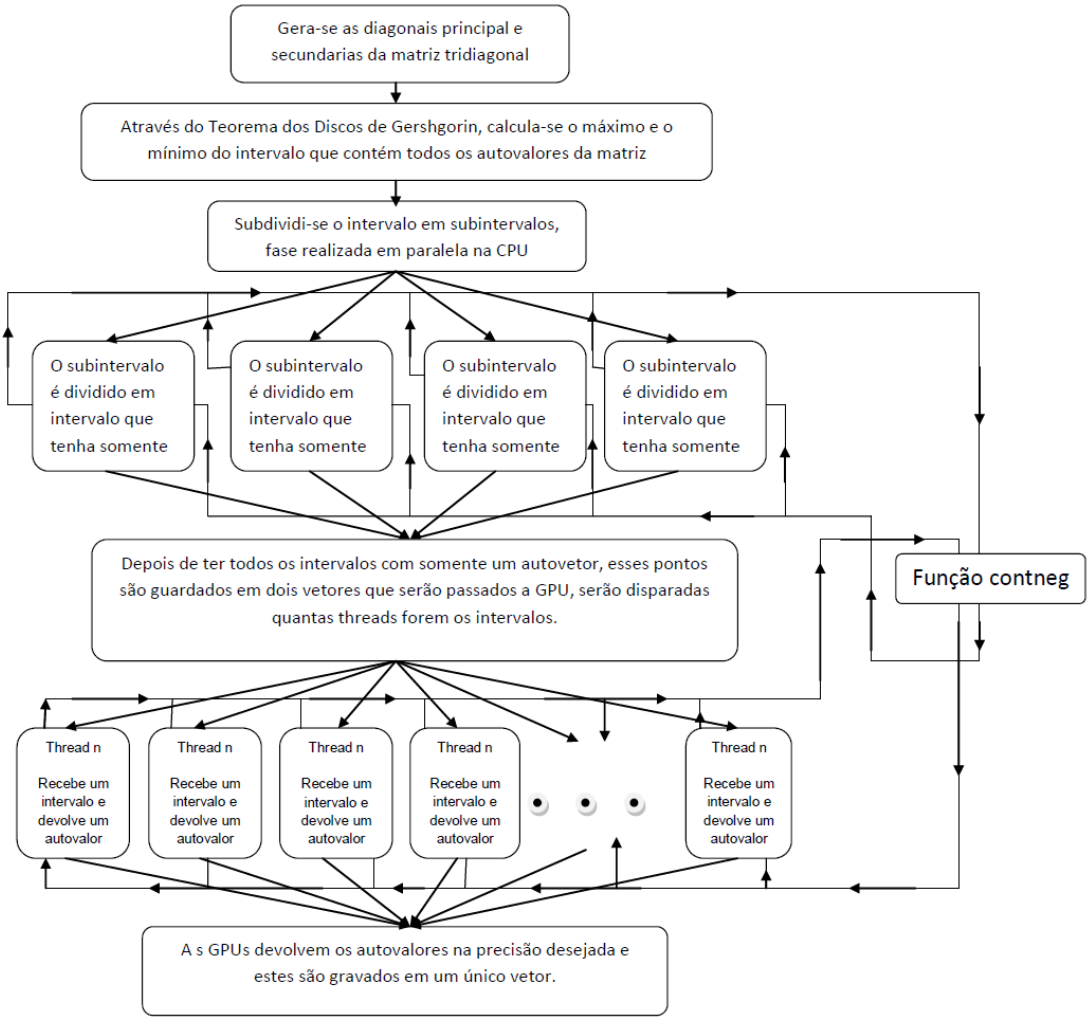

FIGURA 13 – Fluxograma do Programa.

4 Resultados

4.1 Análises dos Tempos Medidos

A máquina utilizada para o cálculos é dotada de dois processadores Intel(R) Xeon(R) CPU E5645 2.40GHz, de 12 núcleos cada, com 94.4 GiB de memória ram e quatro placas gráficas NVIDIA Corporation GF110GL [Tesla C2050 / C2075].

Existem diversas bibliotecas numéricas que implementam, em várias linguagens de programação, o método da bissecção para o cálculo de autovalores. A *linear álgebra package* (LAPACK) e a EISPACK [30], são duas destas bibliotecas, a LAPACK em C e FORTRAN e a EISPACK em FORTRAN.

Neste trabalho, escolhemos para comparação, a rotina DSTEBZ da biblioteca LAPACK. A rotina DSTEBZ realiza o cálculo de autovalores usando o método da bisseccão, mesmo método que usaremos em linguagem C e em CUDA.

Temos na Tabela 3, os tempos, em segundos, gastos para o cálculo dos autovalores da matriz quadrada triagonal de ordem 1024*N(onde N é um multiplicado, a matriz será sempre de 1024 vezes N, com N indo de 1 a 100), em precisão dupla, em LAPACK DSTEBZ (DSTEBZ), paralelizado no processador (MCPU), híbrido, isolamento paralelizado no processador e extração em uma das placas gráficas (GPU) e híbrido, isolamento paralelizado no processador e extração paralelizado nas quatro placas gráficas (MGPU):

Tabela 3 – Tempos de execução do método da bissecção em DSTEBZ, MCPU, GPU e MGPU.

N	DSTEBZ	MCPU	GPU	MGPU
1	0,587	0,069	0,888	0,044
20	179,379	12,792	10,713	6,568
40	700,786	48,982	38,795	26,918
60	1681,930	108,017	82,831	31,763
80	2925,316	189,946	145,069	62,321
100	4334,880	292,558	222,784	90,080

A Figura 14 mostra graficamente as diferenças de tempo. No qual podemos observar que já se tem ganho de tempo com o método em C puro.

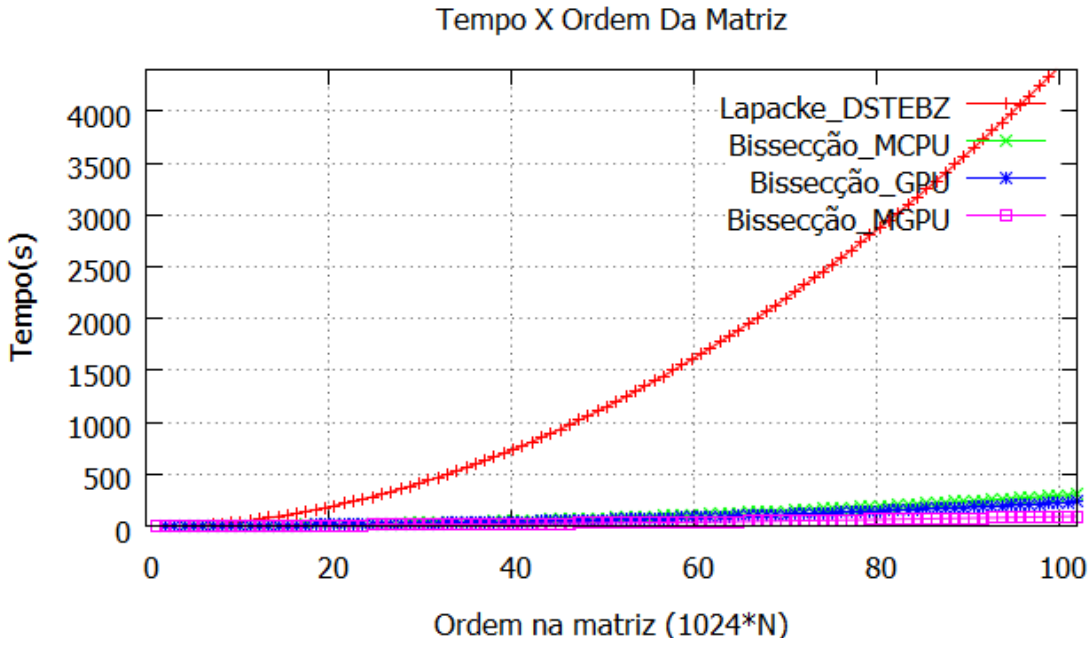

FIGURA 14 – Tempos de execução, em segundos, pela ordem da matriz 1024*N.

A aceleração, definida como a quantidade de vezes que o programa foi mais rápido, que obtivemos entre o DSTEBZ e o MGPU, Tabela 4, foi de:

TABELA 4 – Aceleração obtida em MGPU em relação ao DSTEBZ.

N	DSTEBZ	MGPU	Aceleração
20	179,379	6,568	27,311
60	1681,930	31,763	52,952
100	4334,880	90,080	48,122

Já a aceleração entre o MCPU e o MGPU, Tabela 5, foi de:

Já na Figura 15 na página seguinte visualizamos os tempos gastos entre os nossos métodos, em MCPU e GPU e MGPU.

Os métodos MCPU, GPU e MGPU, compartilham a mesma forma de isolamento, paralelizado em C, se diferenciando na forma de extração, paralelizada na CPU, em uma GPU e nas quatro GPUs respectivamente, na Tabela 6, temos os tempos gasto na fase de extração em segundos.

Tabela 5 – Aceleração obtida em MGPU em relação a MCPU.

N	MCPU	MGPU	Aceleração
20	12,792	6,568	1,947
60	108,017	31,763	3,400
100	292,558	90,080	3,247

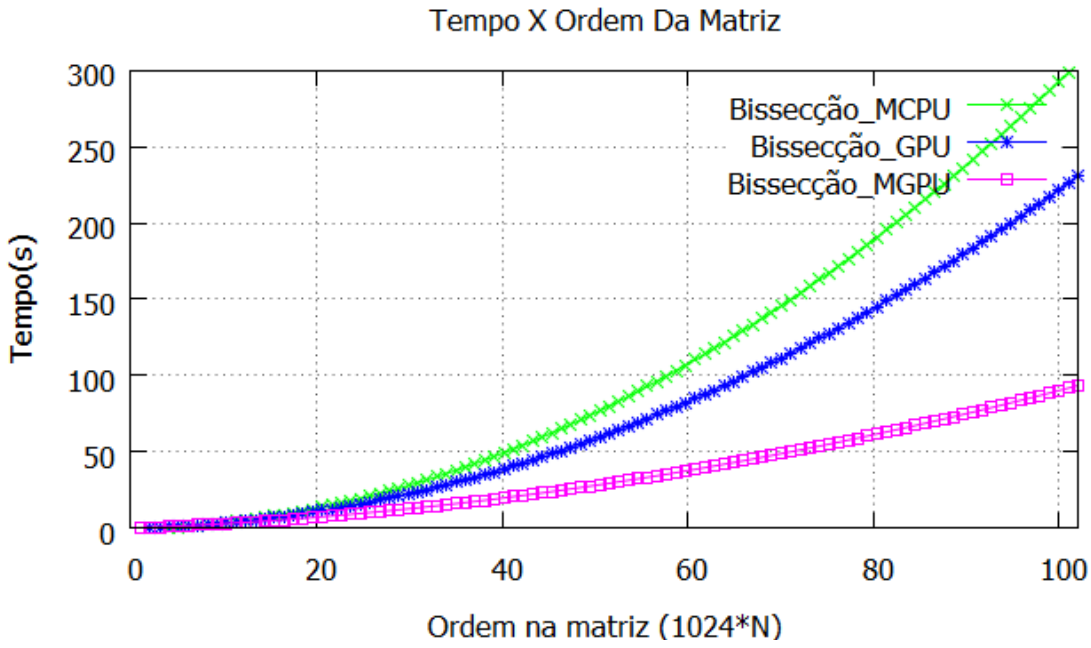

Figura 15 – Tempos de execução, em segundos, pela ordem da matriz 1024*N.

Tabela 6 – Tempos de execusão, da fase de extração, em segundos, em MCPU, GPU e MGPU.

N	MCPU	GPU	MGPU
1	0,042	0,114	0,119
20	11,698	8,822	4,155
40	45,104	33,405	16,567
60	99,672	71,919	19,641
80	175,459	126,516	37,005
100	270,117	194,552	65,620

Já na fase de extração entre, o MCPU e o MGPU, Tabela 7, o MGPU foi em torno de 5 vezes mais rápido, para uma matriz de 1024*N com N igual a 60.

Tabela 7 – Aceleração da fase de extração, em segundos, entre MCPU e MGPU.

N	MCPU	MGPU	Aceleração
20	11,698	4,155	2,815
60	99,672	19,641	5,075
100	270,117	65,620	4,116

Em relação a rotina DSTEBZ, tanto o método paralelizado na CPU, como GPU e MGPU, se mostraram mais eficientes, tendo o MGPU, obtido um ganho de velocidade da ordem de 48.122, para a matriz com N igual a 100.

Comparando a fase de extração entre os métodos de MCPU e MGPU, temos que a aceleração obtida foi da ordem de 4,11 para a matriz com N igual a 100.

4.2 Trabalhos Futuros

Cálculo dos autovetores usando CUDA.

Melhoria na forma de dividir os intervalos na fase de isolamento, de modo que cada CUDA thread realize o mesmo trabalho.

Estender o nosso método para outros tipo de matrizes.

Referências Bibliográficas

1 KIRK, D. B.; WEN-MEI, W. H. *Programming massively parallel processors: a hands-on approach*. [S.l.]: Morgan Kaufmann, 2010.

2 NVIDIA, C. *C Programming Guide (2012)*. [S.l.]: Version.

3 KROEMER, H. Nobel lecture: Quasielectric fields and band offsets: teaching electrons new tricks. *Rev. Mod. Phys.*, v. 73, n. 3, p. 783–793, 2001.

4 NETO, B. G. E. Efeito de campos elétricos, dopagens não-abruptas e interfaces graduais na estrutura eletrônica de poços quânticos de gaas/algaas e gan/algan. 2007.

5 LESSIG, C. Eigenvalue computation with cuda. *NVIDIA techreport*, 2007.

6 NVIDIA. *CUDA*. 2013. [Online; accessed Fevereiro-2013]. Disponível em: <http://www.nvidia.com.br/object/cuda_home_new_br.html>.

7 NVIDIA, C. Nvidiaâs next generation cuda compute architecture: Fermi. *Computer system*, v. 26, p. 63–72, 2009.

8 BEYOND, D. *Beyond 3D. Nvidia g80: Architecture and gpu analysis. @ONLINE*. 2013. [Online; accessed Julho-2013]. Disponível em: <http://www.beyond3d-.com/content/reviews/1>.

9 BEYOND, D. *Beyond 3D. Nvidia gt200: Architecture and gpu analysis. @ONLINE*. 2013. [Online; accessed Julho-2013]. Disponível em: <http://www.beyond3d-.com/content/reviews/51>.

10 NEGRUT, D.; LAMB, D.; GORSICH, D. A high performance computing framework for physics-based modeling and simulation of military ground vehicles. In: INTERNATIONAL SOCIETY FOR OPTICS AND PHOTONICS. *SPIE Defense, Security, and Sensing*. [S.l.], 2011. p. 806002–806002.

11 YAMAZAKI, T.; IGARASHI, J. Realtime cerebellum: A large-scale spiking network model of the cerebellum that runs in realtime using a graphics processing unit. *Neural Networks*, Elsevier, 2013.

12 BARD, D. et al. Cosmological calculations on the gpu. *arXiv preprint arXiv:1208.3658*, 2012.

13 NVIDIA. *cuFFt. @ONLINE*. 2014. [Online; accessed Março-2014]. Disponível em: <https://developer.nvidia.com/cufft>.

14 NVIDIA. *cuBLAS. @ONLINE*. 2014. [Online; accessed Março-2014]. Disponível em: <https://developer.nvidia.com/cublas>.

15 NVIDIA. *cuBLAS-XT. @ONLINE*. 2014. [Online; accessed Março-2014]. Disponível em: <https://developer.nvidia.com/cublasxt>.

16 CUSPARSE. *cuSPARSE. @ONLINE*. 2014. [Online; accessed Março-2014]. Disponível em: <https://developer.nvidia.com/cusparse>.

17 NVIDIA. *CULA. @ONLINE*. 2014. [Online; accessed Março-2014]. Disponível em: <https://developer.nvidia.com/em-photonics-cula-tools>.

18 MAGMA. *MAGMA. @ONLINE*. 2014. [Online; accessed Março-2014]. Disponível em: <http://icl.cs.utk.edu/magma/software/index.html>.

19 LIMA, P. E. de; SOUZA, L. d. F. R. de. Autovalores e autovetores: Conceitos e uma aplicação a um sistema dinâmico. *Revista Eletrônica de Educação e Ciência*, v. 3, n. 1, p. 22–28, 2013.

20 MOURA, C. A. d. Autovalores, estimativas a priori, gershgorin e seus cÃrculos. 2004.

21 GOLUB, G. H.; LOAN, C. F. V. *Matrix Computations Johns Hopkins University Press*. [S.l.]: Baltimore, London, 1989.

22 STURM, J. C. F. MÃ©moire sur la rÃ©solution des Ãquations numÃ©riques. 1829.

23 WILKINSON, J. H.; WILKINSON, J. H.; WILKINSON, J. H. *The algebraic eigenvalue problem*. [S.l.]: Clarendon Press Oxford, 1965.

24 GIVENS, W. A method of computing eigenvalues and eigenvectors suggested by classical results on symmetric matrices. *Nat. Bur. Standards Appl. Math. Ser*, v. 29, p. 117–122, 1953.

25 GIVENS, W. *Numerical computation of the characteristic values of a real symmetric matrix*. [S.l.], 1954.

26 BARTH, W.; MARTIN, R.; WILKINSON, J. Calculation of the eigenvalues of a symmetric tridiagonal matrix by the method of bisection. *Numerische Mathematik*, Springer, v. 9, n. 5, p. 386–393, 1967.

27 KAHAN, W. *Accurate eigenvalues of a symmetric tri-diagonal matrix*. [S.l.], 1966.

28 KAHAN, W.; VARAH, J. *TWO WORKING ALGORITHMS FOR THE EIGENVALUES OF A SYMMETRIC TRIDIAGONAL MATRIX*. [S.l.], 1966.

29 WILKINSON, J. Calculation of the eigenvalues of a symmetric tridiagonal matrix by the method of bisection. *Numerische Mathematik*, Springer, v. 4, n. 1, p. 362–367, 1962.

30 NAT, L. A. *EISPACK. Eigensystem package*. [S.l.]: Illinois, 1972.

31 BERNSTEIN, H. J. An accelerated bisection method for the calculation of eigenvalues of a symmetric tridiagonal matrix. *Numerische Mathematik*, Springer, v. 43, n. 1, p. 153–160, 1984.

32 BADÍA, J. *Algoritmos Paralelos para el Cálculo de los Valores Propios de Matrices Estructuradas*. Tese (Doutorado) — Ph. D. Thesis, Univ. Politécnica de Valencia, 1996.

33 RALHA, R. Parallel solution of the symmetric tridiagonal eigenvalue problem on a transputer network. *SEMNI*, v. 93, p. 1026–1033, 1993.

APÊNDICE A – Glossário De Termos Em Inglês

thread – Linha ou Encadeamento de execução de um programa.

Memoria cache — Na área da computação, cache é um dispositivo de acesso rápido, interno a um sistema, que serve de intermediário entre um operador de um processo e o dispositivo de armazenamento ao qual esse operador acede. A vantagem principal na utilização de um cache consiste em evitar o acesso ao dispositivo de armazenamento - que pode ser demorado -, armazenando os dados em meios de acesso mais rápidos.

memory bandwidth — Largura de banda de memória – é a taxa à qual os dados podem ser lidos ou armazenados em uma memória de semicondutor por um processador . Largura de banda de memória é normalmente expressa em unidades de bytes / segundo , embora isso possa variar para sistemas com tamanhos de dados naturais que não são um múltiplo dos bytes de 8 - bit vulgarmente utilizados .

multicore trajectory – Conceito que consiste em fazer com que programa sequênciais sejam executados movendo-os entre os múltiplos núcleos do processador.

hyperthreading – tecnologia criada pela Intel, na qual o microprocessador simula ter mais núcleos, e maximiza a velocidade de execução dos programas sequênciais.

many-core trajectory – Conceito que consiste emu fazer o programa rodar em vários núcleos ao mesmo tempo.

DRAM – é um tipo de memória RAM de acesso direto que armazena cada bit de dados num condensador ou capacitor. O número de elétrons armazenados no condensador determina se o bit é considerado 1 ou 0.

multithread – sistemas que suportam múltiplas threads.

benchmark – é o ato de executar um programa de computador, um conjunto

de programas ou outras operações, a fim de avaliar a performance relativa de um objeto, normalmente executando uma série de testes padrões e ensaios nele.

overflow/underflow – A condição de overflow ou underflow ocorre quando o valor atribuído a uma variável é maior que o maior valor ou menor que o menor valor que o tipo desta variável consegue representar.