UNIVERSIDADE FEDERAL DA PARAÍBA
CENTRO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA

Paralelizando o MOPAC usando CUDA e bibliotecas de Matrizes Esparsas

Carlos Peixoto Mangueira Júnior

JOÃO PESSOA-PB
Março-2012

UNIVERSIDADE FEDERAL DA PARAÍBA
CENTRO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA

Paralelizando o MOPAC usando CUDA e Bibliotecas de Matrizes Esparsas

CARLOS PEIXOTO MANGUEIRA JÚNIOR

JOÃO PESSOA-PB
Março-2012
II

CARLOS PEIXOTO MANGUEIRA JÚNIOR

Paralelizando o MOPAC usando CUDA e Bibliotecas de Matrizes Esparsas

DISSERTAÇÃO APRESENTADA AO CENTRO DE INFORMÁTICA DA UNIVERSIDADE FEDERAL DA PARAÍBA, COMO REQUISITO PARCIAL PARA OBTENÇÃO DO TÍTULO DE MESTRE EM INFORMÁTICA (SISTEMAS DE COMPUTAÇÃO).

Orientador: Prof. Dr. Lucídio dos Anjos Formiga Cabral

Co-orientador: Prof. Dr. Gerd Bruno da Rocha

JOÃO PESSOA-PB

Março-2012

Carlos Peixoto Mangueira Júnior

Paralelizando o MOPAC usando CUDA e bibliotecas de Matrizes Esparsas

Esta Dissertação foi julgada adequada para obtenção do Título de Mestre em Informática e aprovada em sua forma final pelo Programa de Pós-Graduação em Informática da Universidade Federal da Paraíba.

João Pessoa, 23/03/2012

_____

Tatiana Aires Tavares, Dra.
Coordenadora do Curso

Banca Examinadora:

_____

Lucídio dos Anjos Formiga Cabral, Dr. (Orientador)
Doutor em Engenharia de Sistemas e Computação (UFRJ)

_____

Gerd Bruno da Rocha, Dr. (Co-orientador)
Doutor em Química (UFPE)

_____

Luiz Satoru Ochi, Dr.
Doutor em Engenharia de Sistemas e Computação (UFRJ)

_____

Jairo Rocha de Faria, Dr.
Doutor em Modelagem Computacional (LNCC)

Dedico este trabalho a cada um que faz parte da minha história

Uma longa viagem começa com um único passo

Lao-Tsé

RESUMO

Este trabalho apresenta a implementação de algoritmos paralelos cujo objetivo principal é acelerar a execução de cálculos numéricos existentes em programas de Química Quântica. Estes programas utilizam alguns métodos cuja ordem de complexidade varia entre $O(n^3)$ e $O(n^5)$, onde o parâmetro n está relacionado à quantidade de átomos de uma molécula. Isto se torna um fator limitante quando se quer trabalhar com sistemas moleculares contendo milhares de átomos, como por exemplo, proteínas, DNA e polissacarídeos. É explorado tanto o paralelismo proporcionado pelas placas gráficas e pelo modelo de programação CUDA como também são utilizadas bibliotecas para manipulação de matrizes esparsas, que são comuns nestes cálculos. Os resultados obtidos demonstram ganhos superiores a 100% para as instâncias testes.

ABSTRACT

This work describes the implementation of parallel algorithms whose main goal is to accelerate the implementation of numerical calculations existing in quantum chemistry programs. These programs use some methods whose order of complexity varies from $O(n^3)$ and $O(n^5)$, where n is the parameter related to the amount of atoms in a molecule. This becomes a limiting factor when one wants to work with molecular systems containing thousands of atoms, such as proteins, DNA and polysaccharides. It is explored both the parallelism provided by graphics cards and the CUDA programming model are also used libraries for manipulating sparse matrices, which are common in these calculations. The results show gains of more than 100% for test instances.

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

A Ciência da Computação é uma área das Ciências Exatas que está em constante e crescente desenvolvimento e que se preocupa principalmente em estudar os algoritmos, suas aplicações e sua implementação, na forma de software, para execução em computadores eletrônicos. Muitos destes algoritmos são usados nas mais diversas áreas das ciências como: Matemática, Física, Química, entre outras. O uso de computadores se tornou essencial para ajudar na resolução de problemas nos domínios destas áreas, já que *hardwares* e *softwares* mais sofisticados estão sendo desenvolvidos para facilitar o avanço das pesquisas.

Neste trabalho, em especial, será apresentado algumas das pesquisas que estão sendo desenvolvidas na área de Química Quântica Computacional. A Química é a ciência que trata com a construção, transformação e propriedades das moléculas. A Química Teórica é um campo da Química que trata com modelos e métodos matemáticos que são combinados com as leis fundamentais da Física para estudar os processos químicos de relevância.

Moléculas são tradicionalmente consideradas como sendo compostas de átomos, ou num senso mais geral, como uma coleção de partículas com cargas: núcleos positivos e elétrons negativos. A força física mais importante para os fenômenos químicos é a interação de Coulomb entre estas partículas carregadas. As moléculas são diferentes uma das outras porque contém núcleos diferentes e número diferentes de elétrons, ou os centros nucleares podem estar numa forma geométrica diferente.

Dado um conjunto de núcleos e elétrons que compõem um átomo, a Química Teórica pode calcular:

- O arranjo geométrico dos núcleos que corresponde a uma molécula estável;
- Suas energias relativas;
- Suas propriedades (momento dipolo, polarizabilidade, etc...);
- A taxa pela qual uma molécula estável pode se transformar em outra molécula;
- A dependência de tempo de estruturas moleculares e propriedades.

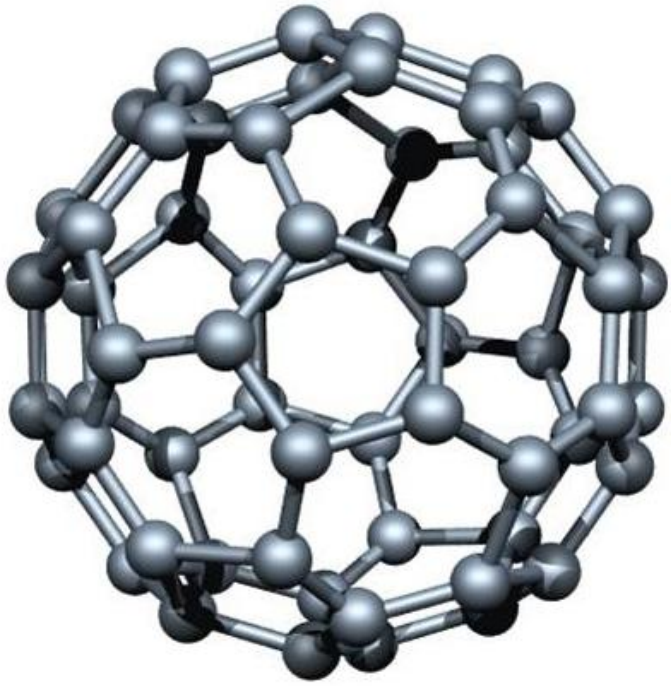

Figura 1: Fulereno - C60

De acordo com (RUBENSSON, RUDBERG e SALEK, 2007) e (THIEL, 2003), as pesquisas da Química Quântica Computacional estão direcionadas a desenvolver métodos eficientes para serem aplicados em sistemas moleculares considerados de grande dimensão, pois as equações matemáticas presentes nestes métodos possuem ordens de complexidade variando entre $O(n^3)$ até $O(n^5)$, onde o parâmetro *n* está relacionado à quantidade de átomos de uma molécula. Ou seja, a aplicação destes métodos a sistemas moleculares com milhares de átomos - como proteínas, polissacarídeos, DNA e polímeros - fica limitada pelo tempo de execução.

Desenvolver métodos químicos de escalonamento linear, cujas ordens de complexidade fiquem próximas a $O(n)$, consiste em um desafio que possibilita o estudo de moléculas que antes não foram estudadas por métodos de química quântica. As técnicas de escalonamento linear que estão sendo desenvolvidas têm como objetivo principal a substituição de etapas que são computacionalmente custosas, como é o caso da diagonalização da matriz de Fock e da computação de integrais atômicas, por métodos mais eficientes. Entre estes métodos pode-se citar a Pseudodiagonalização (STEWART, CSÁSZÁR e PULAY, 1982), a técnica CG-DMS (*Conjugate Gradient-Density Matrix Search*) (DANIELS, MILLAM e SCUSERIA, 1997), o método Dividir e Conquistar (VAN DER VAART, GOGONEA, *et al.*, 2000) e a Purificação (PALSER e MANOLOPOULOS, 1998). Existem

vários estudos que comparam a eficiência destes métodos como é observado em (RUDBERG e RUBENSSON, 2011) e (DANIELS e SCUSERIA, 1999).

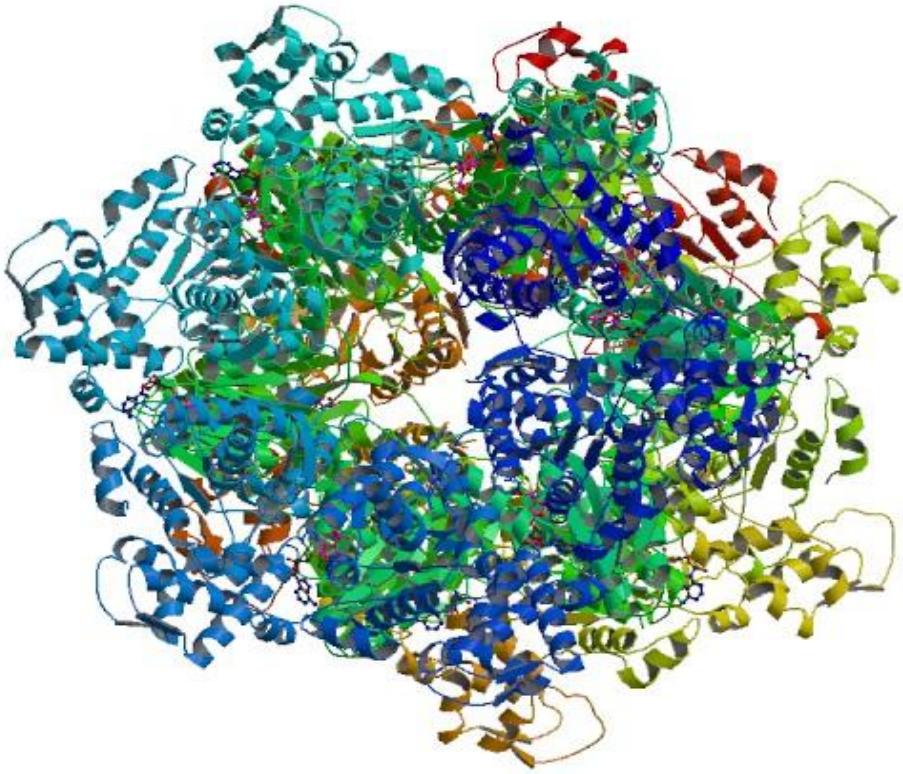

Figura 2: Complexo Protease-Chaperona - PDB: 13GI

Os métodos de Química Quântica podem ser classificados em métodos *ab initio* ou semi-empíricos. Os métodos semi-empíricos surgiram como uma das maneiras encontradas para a redução do tempo computacional gasto para o tratamento de sistemas moleculares na obtenção de propriedades que exigem um maior esforço computacional como: propriedades termodinâmicas, cinéticas, cálculos de superfície de potencial, entre outras.

Uma característica observada na realização de cálculos semi-empíricos é o fato de que algumas matrizes que surgem nos cálculos de sistemas "não-metálicos" possuem uma grande quantidade de elementos desprezíveis que crescem linearmente em relação ao tamanho do sistema, como se observa em estudos realizados por (MASLEN, OCHSENFELD, *et al.*, 1998) e (MILLAM e SCUSERIA, 1997). O esforço computacional pode, então, ser reduzido caso se evite armazenar tais elementos e se evitem operações sobre elementos redundantes das matrizes. Matrizes com uma grande quantidade de elementos desprezíveis são consideradas

esparsas (PISSANETSKY, 1984) e elas são de grande importância nestes cálculos. Para se beneficiar destas características de esparsidade é preciso utilizar formatos diferenciados para a realização de operações sobre estas matrizes. Entre os formatos mais conhecidos pode-se citar o CSR (*Compressed Sparse Row*) ou o CSC (*Compressed Sparse Column*) que serão detalhados nas próximas seções. Em (RUBENSSON, RUDBERG e SALEK, 2007) é, ainda, proposto um esquema de hierarquia de matrizes esparsas que pode ser utilizado para acelerar estes métodos.

Na comunidade, é encontrada uma variedade de programas de Química Quântica, dentre os quais estão: o MOPAC (Stewart Computational Chemistry - MOPAC - Home Page) e o GAUSSIAN (Official Gaussian Website) que possuem os mais variados métodos para o tratamento de moléculas. Entretanto, alguns destes programas foram desenvolvidos inicialmente para serem executados em máquinas de um único processador e por isso eles também estão limitados a moléculas com uma quantidade menor de átomos. Entre estes programas vale destacar que o MOPAC é um pacote semi-empírico de propósito geral e de código aberto para estudo de estruturas moleculares e, também, é largamente utilizado na comunidade científica.

Entre algumas das técnicas computacionais existentes responsáveis pela aceleração do tempo de execução de um programa está a paralelização. A paralelização melhora o tempo de execução do programa através da distribuição da carga de trabalho entre um conjunto de nós de computação. Encontram-se na literatura, trabalhos cujo intuito é paralelizar alguns dos programas químicos citados anteriormente como se vê em (LIN, 2000), em (NEEDHAM, 2010) e em (DE JONG, BYLASKA, *et al.*, 2010). Porém, paralelizar estes programas não é uma tarefa muito fácil, pois a paralelização destes programas requer um conhecimento do campo de atuação destes programas, que nestes casos é o campo da Química, o que requer um esforço maior do cientista da computação para entender estes novos conceitos.

Para se utilizar a paralelização é preciso usar computadores paralelos como os computadores com processadores *multicore* ou os *clusters*. Como a velocidade do *clock* dos processadores atuais está limitada por alguns fatores como a dissipação de calor, uma atenção especial está sendo direcionada às GPUs (*Graphic Unit Processor*). As GPUs se apresentam como uma alternativa para programação massiva paralela de dados, pois sua arquitetura foi desenhada para garantir altas taxas de processamento de vídeos com a realização de um grande número de cálculos de ponto-flutuante por *frame* do vídeo. Outra vantagem do *hardware* da GPU é a presença de uma grande quantidade de *threads* de execução que são organizadas em *grids* computacionais. A NVIDIA (NVIDIA Home) com o objetivo de se

beneficiar destas características de alto processamento paralelo das GPUs apresentou em 2007 a arquitetura CUDA *(Computer Unified Device Architecture)* (What is CUDA | NVIDIA Developer Zone) que incluiu vários novos componentes projetados estritamente para a computação em GPUs e com o objetivo de amenizar muitas das limitações que impediam as GPUs anteriores de serem úteis para a computação de propósito geral.

Este novo paradigma de programação, que é a programação para GPUs, possibilita a programação para sistemas heterogêneos compostos por CPU e GPU através da utilização de extensões para a linguagem C, o que facilita mais ainda o trabalho dos programadores. Programas desenvolvidos para a arquitetura CUDA podem alcançar um *speedup* em suas aplicações de 10 a 100 vezes.

No site da NVIDIA tem uma seção chamada de *Computational Chemistry* (Computational Chemistry) onde podem ser encontrados programas e trabalhos de Química Quântica que se beneficiam do poder das GPUs.

1.1 Objetivos do Trabalho

Diante do cenário exposto, este trabalho tem como objetivo principal desenvolver versões paralelas dos métodos que substituem a diagonalização da matriz de Fock, explorando tanto o paralelismo massivo de dados oferecidos pelas placas gráficas com arquitetura CUDA, como as possibilidades de manipulação de matrizes esparsas com o intuito de acelerar algumas atividades realizadas no campo da Química Quântica. Estas versões serão adequadas e integradas no MOPAC que é um dos principais programas da Química Quântica.

Entre os objetivos específicos pode-se citar o estudo e a aplicação de bibliotecas apropriadas para tratamento de matrizes esparsas e o desenvolvimento de algumas operações básicas de álgebra linear em CUDA para serem usadas nas técnicas de escalonamento linear. Os tempos de execução dos métodos paralelos desenvolvidos serão comparados com os tempos de execução dos métodos seriais executados na CPU com a utilização de cenários reais.

1.2 Organização da Dissertação

Esta dissertação de mestrado está organizada da seguinte maneira:

- Capítulo 2 – Fundamentação Teórica: apresentação dos conceitos necessários para o desenvolvimento desta pesquisa;
- Capítulo 3 – Método Proposto: apresentação dos métodos, técnicas e ferramentas utilizados;
- Capítulo 4 – Resultados e Discussão: apresentação dos resultados e da discussão da aplicação do código desenvolvido mostrando um comparativo com os resultados já existentes na literatura;
- Conclusão: apresentação das conclusões a partir da pesquisa que está sendo realizada.

2 FUNDAMENTAÇÃO TEÓRICA

Nesta seção serão apresentados alguns conceitos necessários para a consolidação desta pesquisa. Serão mostrados os principais conceitos envolvendo Química Quântica, Matrizes Esparsas, Técnicas de escalonamento linear, o programa MOPAC e o paradigma de programação em CUDA.

2.1 Métodos dos Orbitais Moleculares

Em suma, existem duas classes de métodos teóricos para o estudo de moléculas: aqueles que se baseiam na Química Quântica e aqueles que usam modelos empíricos baseados na Física Clássica para descrever estruturas moleculares, conhecidos como mecânica molecular. Os métodos de Química Quântica podem ainda ser divididos em três categorias: *ab initio*, semi-empírico e DFT (*Density Functional Theory*).

Os métodos de Química Quântica são mais exatos e por isso demandam maior tempo de execução, enquanto que, os métodos que usam modelos clássicos são menos exatos, contudo, suas equações podem ser resolvidas muito mais rapidamente.

Para que os métodos quânticos sejam viáveis para moléculas com milhares de átomos existem algumas alternativas. Uma alternativa são os métodos QM/MM que combinam a mecânica quântica com a mecânica molecular. Estes métodos particionam um sistema grande em pequenas regiões onde os eventos eletrônicos ocorrem e em um ambiente que influencia estes eventos. Essas pequenas regiões podem ser o sítio ativo de uma proteína (macromolécula com milhares de átomos), assim, o sítio ativo pode ser tratado com um nível apropriado de mecânica quântica, enquanto que, o ambiente é descrito no nível de mecânica molecular.

A segunda alternativa é alcançar o escalonamento linear do esforço computacional dos cálculos de mecânica quântica puros. Esta alternativa se baseia no conceito de localidade da densidade eletrônica, pois a maioria das interações químicas relevantes atua em pequenas regiões e, portanto, podem ser descritas por algoritmos de ordem de complexidade $O(n)$ que devem explorar a localidade através de truncamentos, enquanto que, as interações

eletrostáticas de atuação de longo alcance devem ser tratadas separadamente por técnicas especiais.

Os métodos da mecânica quântica se propõem a resolver a equação de Schrödinger que está descrita na Equação 1:

$$H_{el}\Psi_{el} = E_{el}\Psi_{el}. \quad (1)$$

Na Equação 1, H_{el} representa o Hamiltoniano, Ψ_{el} representa a função de onda e E_{el} representa a energia eletrônica. Na aproximação orbital, a função de onda de n-elétrons é representada como um determinante de *Slater* anti-simétrico construído a partir dos orbitais de um-elétron que são determinados de modo variacional. No caso de moléculas, os orbitais ψ_i são normalmente escritos como uma combinação linear dos orbitais atômicos φ_μ como se vê na Equação 2:

$$\psi_i = \sum_\mu C_{\mu i}\varphi_\mu. \quad (2)$$

A minimização variacional da energia eletrônica E_{el} com respeito aos coeficientes $C_{\mu i}$ leva à equação de Roothaan-Hall que está descrita na Equação 3:

$$FC = SCE. \quad (3)$$

Na Equação 3, C denota a matriz coeficiente, E denota a matriz diagonal contendo os orbitais de energia ε_i e S é a matriz de *overlap*.

Usando a notação padrão e as unidades atômicas, os elementos da matriz de Fock para sistemas *closed-shell* são dados pela Equação 4 e Equação 5:

$$F_{\mu v} = H_{\mu v} + G_{\mu v}, \quad (4)$$

$$G_{\mu v} = \sum_\lambda \sum_\sigma P_{\lambda\sigma}\left[< \mu v \left| \lambda\sigma > - \frac{1}{2} < \mu\lambda \right| v\sigma >\right]. \quad (5)$$

A integral de um-elétron $H_{\mu v}$ inclui o termo da energia cinética e as atrações núcleo-elétron, enquanto que, as integrais de dois-elétrons $< \mu v|\lambda\sigma >$ representam a repulsão entre

distribuições de cargas correspondentes. Os elementos da matriz densidade $P_{\lambda\sigma}$ são definidos pelo somatório sobre os orbitais ocupados como é mostrado na Equação 6:

$$P_{\lambda\sigma} = 2\sum_{i}^{Nocc} C_{\lambda i}C_{\sigma i}. \quad (6)$$

A energia eletrônica é dada pela Equação 7:

$$E_{el} = \frac{1}{2}\sum_{\mu}\sum_{v} P_{\mu v}(H_{\mu v} + F_{\mu v}). \quad (7)$$

A energia total é obtida através da adição da energia de repulsão nuclear mais a energia eletrônica.

Da teoria *ab initio* podem-se deduzir algumas coisas. Primeiro, que para *N* funções de base, há $O(N^4)$ integrais de dois-elétrons a serem calculadas e o esforço computacional terá ordem de complexidade de $O(N^4)$. Segundo, que cada iteração SCF envolve a resolução de um problema de autovalor generalizado, tipicamente através de uma diagonalização cujo esforço computacional é da ordem de $O(N^3)$.

Os métodos semi-empíricos introduzem aproximações nas integrais para desprezar a maioria das integrais atômicas, cujos valores são praticamente nulos, que aparecem no formalismo *ab initio*. Para compensar o erro causado por estas aproximações, as integrais restantes são descritas por expressões paramétricas. A equação secular nestes métodos assume uma forma simplificada como a Equação 8:

$$FC = CE. \quad (8)$$

Porém, a solução iterativa, que corresponde à diagonalização, ainda tem ordem de complexidade de $O(N^3)$. Por outro lado, o cálculo das integrais passam a ter ordem de complexidade de $O(N^2)$.

A equação secular tem que ser resolvida iterativamente e, portanto, uma abordagem convencional do SCF-MO envolve os seguintes passos:

1. Cálculo das integrais de um-elétron e dois-elétrons;
2. Chute inicial para a matriz de densidade;
3. Formação da matriz de Fock;
4. Diagonalização da matriz de Fock;
5. Formação de uma nova matriz de densidade;
6. Checagem de convergência.

Estes passos podem estão representados pelo seguinte Diagrama de Atividades que é mostrado na Figura 3:

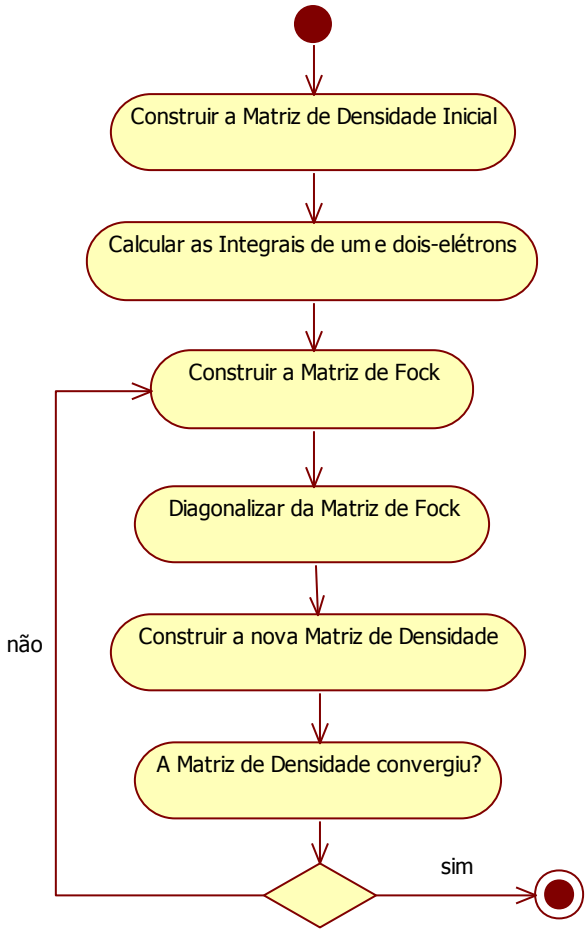

Figura 3 Fluxo SCF-MO

Nos cálculos convencionais do SCF-MO são usados algumas técnicas para auxiliar e acelerar a convergência como: *Extrapolation*, *Damping*, *Level Shifting* e *Direct inversion in the iterative subspace – DIIS*.

Na técnica *Extrapolation* a ideia é formar uma nova matriz de Fock, durante passos selecionados do SCF, não apenas a partir da matriz de densidade corrente, mas de uma matriz de densidade modificada pela extrapolação das anteriores. Uma prática comum é usar, no máximo, três matrizes de densidade anteriores para extrapolação.

A técnica *Damping* tem como objetivo eliminar problemas causados por oscilações. Se a matriz de densidade da iteração *k* tende a ser muito próxima da matriz da iteração *k+2* e um pouco diferente da matriz da iteração *k+1*, então a variação pode ser amenizada, durante a formação da matriz de Fock, utilizando-se a combinação convexa: $\propto * P_k + (1-\propto) * P_{k+1}$, na qual o parâmetro α pertence ao intervalo $(0,1)$, ao invés da matriz de densidade corrente. O parâmetro $\propto$ pode tanto ser constante como pode diminuir dinamicamente durante o procedimento SCF.

Entre duas iterações SCF há uma mistura de orbitais compostos pelos orbitais ocupados e virtuais. Problemas de convergência podem surgir se esta mistura for tão nítida causando oscilações. A técnica *Level Shifting* aumenta a energia dos orbitais virtuais e, portanto, reduz esta mistura. Isto fará com que as iterações SCF sejam mais suaves e assim auxiliará a convergência em casos difíceis. Mas esta técnica pode reduzir, porém, a taxa de convergência em casos que não são críticos.

A técnica *DIIS* (HAMILTON e PULAY, 1986) é uma técnica de extrapolação que acelera a convergência e mais ainda, pode levar a convergência mesmo em casos difíceis onde outros procedimentos falham. Seu foco está na sequência de matrizes de Fock que são formadas durante os ciclos SCF, na qual, uma matriz de erro é calculada a cada iteração ($FPS - SPF$ nos métodos *ab initio* e $FP - PF$ nos métodos semi-empíricos). Por fim é determinada uma combinação linear das matrizes de Fock disponíveis anteriormente que minimizam a função erro através da solução de um sistema de equações lineares apropriado. Esta combinação linear é usada como a próxima matriz de Fock no procedimento SCF. O custo relativo da técnica *DIIS* nos métodos *ab initio* é considerado desprezível, mas é considerado substancial nos métodos semi-empíricos. Portanto, os métodos semi-empíricos normalmente utilizam técnicas mais simples para acelerar a convergência do procedimento SCF, usando a técnica *DIIS* somente se estas técnicas mais simples não funcionarem.

Outra questão importante envolvendo a convergência do método SCF é o chute inicial da matriz de densidade. Claro que um chute inicial de alta qualidade facilitará e acelerará a convergência. Os métodos semi-empíricos são normalmente um pouco mais robustos em relação à convergência do SCF e geralmente iniciam com uma matriz de densidade diagonal com o número correto de elétrons distribuídos de tal forma que os átomos sejam neutros. Um

chute mais sofisticado pode ser obtido a partir da diagonalização da matriz de core Hamiltoniana de um-elétron.

2.2 Métodos de Escalonamento Linear

O cálculo da matriz de densidade de um-elétron para uma dada matriz Hamiltoniana é uma operação de grande importância nos cálculos de estrutura eletrônica em larga escala. Basicamente existem duas categorias de métodos para encontrar a matriz de densidade: um que envolve um problema de minimização e outro que envolve um problema de purificação (expansão polinomial). A escolha de um destes dois métodos irá interferir fortemente nos resultados dos cálculos. As principais diferenças entre estes métodos serão explicadas adiante.

2.2.1 Cálculo da matriz de densidade

O cálculo da matriz de densidade para uma dada matriz Hamiltoniana F na temperatura eletrônica zero pode ser definido através das Equações 9-11:

$$F_{\perp} = Z^T F Z, \tag{9}$$

$$D_{\perp} = \theta(\mu I - F_{\perp}), \tag{10}$$

$$D = Z D_{\perp} Z^T. \tag{11}$$

Na Equação 10, $\theta(x)$ é a *Heaviside Step Function* e μ é chamado o potencial químico, um valor entre os autovalores correspondentes aos orbitais ocupados e não ocupados, respectivamente. A matriz Z é um fator inverso da matriz de *overlap* S tal que $Z^T S Z = I$.

2.2.2 Métodos de minimização

Os métodos de minimização propõem a minimização do funcional apresentado na Equação 12 para computar a matriz de densidade:

$$\Omega(X_\perp) = Tr[(3X_\perp^2 - 2X_\perp^3)(F_\perp - \mu I)]. \quad (12)$$

Existem diversas variantes para este método, e em todas estas variantes são usadas uma busca linear analítica que constitui a parte significante do esforço computacional para este método. Uma característica deste método é que ele não acumula erros durante as iterações conforme é apresentado em (RUDBERG e RUBENSSON, 2011).

2.2.3 CG-DMS

Um exemplo de método de minimização é a técnica CG-DMS (*Conjugate Gradient Density Matriz Search*). Nesta técnica, o equivalente ao comutador da matriz densidade com a matriz de Fock é minimizado usando um algoritmo baseado em gradientes conjugados, aplicando a restrição de que a matriz densidade deva ser idempotente e normalizada. No limite da convergência, a matriz densidade deve comutar com a matriz de Fock e a energia eletrônica deve ser um mínimo. Dessa forma, esta técnica evita a diagonalização da matriz de Fock, escalonando linearmente com o tamanho do sistema.

Para as equações a seguir, considere $Tr(X)$ como sendo o traço da matrix X. A matriz densidade P é uma matriz simétrica ($P = P^T$), calculada a partir da Equação 13, normalizada para o número de elétrons ($Tr(P) = \frac{N_{el}}{2}$) e idempotente ($P^2 = P$). A idempotência da matriz densidade pode ser vista através da relação mostrada na Equação 14. Dessa forma, a idempotência de P surge da condição de ortonormalidade dos autovetores LCAO-MO, C_{occ}. Na Equação 14, I é a matriz identidade de mesma ordem.

$$P = C_{occ}{C^T}_{occ}, \quad (13)$$

$$PP = C_{occ}(C^T{}_{occ}C_{occ})C^T{}_{occ} = C_{occ}(I)C^T{}_{occ} = P. \quad (14)$$

As três condições apresentadas devem ser satisfeitas por qualquer matriz densidade que represente um sistema molecular de camada fechada (molécula onde todos os orbitais moleculares ocupados são duplamente preenchidos com elétrons) e se tornam vínculos (ou restrições) a serem empregados durante a técnica de CG-DMS.

A energia eletrônica E de um sistema molecular camada fechada é dada pela Equação 15 e depende da matriz densidade P:

$$E(P) = Tr[(Hc + F)P]. \quad (15)$$

Na Equação 15, Hc é o Hamiltoniano de um-elétron e F é a matriz de Fock que é a única matriz que possui dependência com a matriz densidade.

A primeira derivada em relação à matriz P é mostrada na Equação 16:

$$\varepsilon = 2Tr[FP]. \quad (16)$$

Isto implica que a solução do problema de autovalores generalizado de Roothaan-Hall é equivalente a otimização da soma das energias dos orbitais dos orbitais moleculares ocupados (sujeita a condição de ortonormalidade dos MOs ocupados).

A condição de normalização pode ser incorporada na Equação 15 de uma maneira direta e para isso utiliza-se um multiplicador de Lagrange denotado por μ. A condição de idempotência pode ser incorporada na Equação 15 utilizando a estratégia de purificação da matriz densidade que é obtida a partir da substituição mostrada na Equação 17 que é também conhecida como purificação de McWeeny:

$$\rho = 3P^2 - 2P^3. \quad (17)$$

Dessa forma o novo funcional final a ser minimizado obedecerá à Equação 18:

$$\Omega(\rho) = Tr[\rho F] + \mu\left(Tr[\rho] - \frac{N_{el}}{2}\right). \quad (18)$$

A minimização desse funcional é o substituto da diagonalização da matriz de Fock e sendo assim é feita a cada iteração SCF.

Qualquer técnica numérica de minimização de funções não-lineares pode ser utilizada para esse propósito. A abordagem CG-DMS utiliza a técnica numérica de gradientes conjugados para encontrar o mínimo de uma função. Para esse método ser executado precisa-se conhecer o gradiente da função a ser minimizada. A expressão que calcula o gradiente do funcional é apresentado na Equação 19:

$$G(P) = -\nabla\Omega(P) = 3(FP + PF) - 2(FP^2 + PFP + P^2F) + \mu I. \quad (19)$$

De uma maneira rápida, o método do gradiente conjugado usado para minimização de funções usa propriedades da matriz Hessiana (H) para gerar uma seqüência de direções de busca ortogonais H_k. São realizadas buscas lineares analíticas para determinar um tamanho ótimo de passo a cada iteração. Esse procedimento gera a cada iteração uma nova matriz densidade que é comparada à matriz de densidade da iteração para verificação dos critérios de convergência. No fim do procedimento, a matriz de densidade resultante irá minimizar o funcional mostrado na Equação 18.

Na Equação 20 é mostrada a expressão usada para descobrir o tamanho ótimo do passo a cada iteração. Trata-se de uma equação do segundo grau em λ, cujos coeficientes estão mostrados nas Equações 21-23.

$$\frac{d\Omega(P + \lambda H)}{d\lambda} = b + 2c\lambda + 3d\lambda^2 = 0, \quad (20)$$

$$b = Tr[HG(P)], \quad (21)$$

$$c = Tr[3HHF - 2HHPF - 2HPHF - 2PHHF], \quad (22)$$

$$d = -2Tr[HHHF]. \quad (23)$$

A Figura 4 mostra um Diagrama de Atividades que representa o algoritmo SCF contendo a técnica CG-DMS e a Figura 5 mostra um Diagrama de Atividades que representa o algoritmo da técnica CG-DMS.

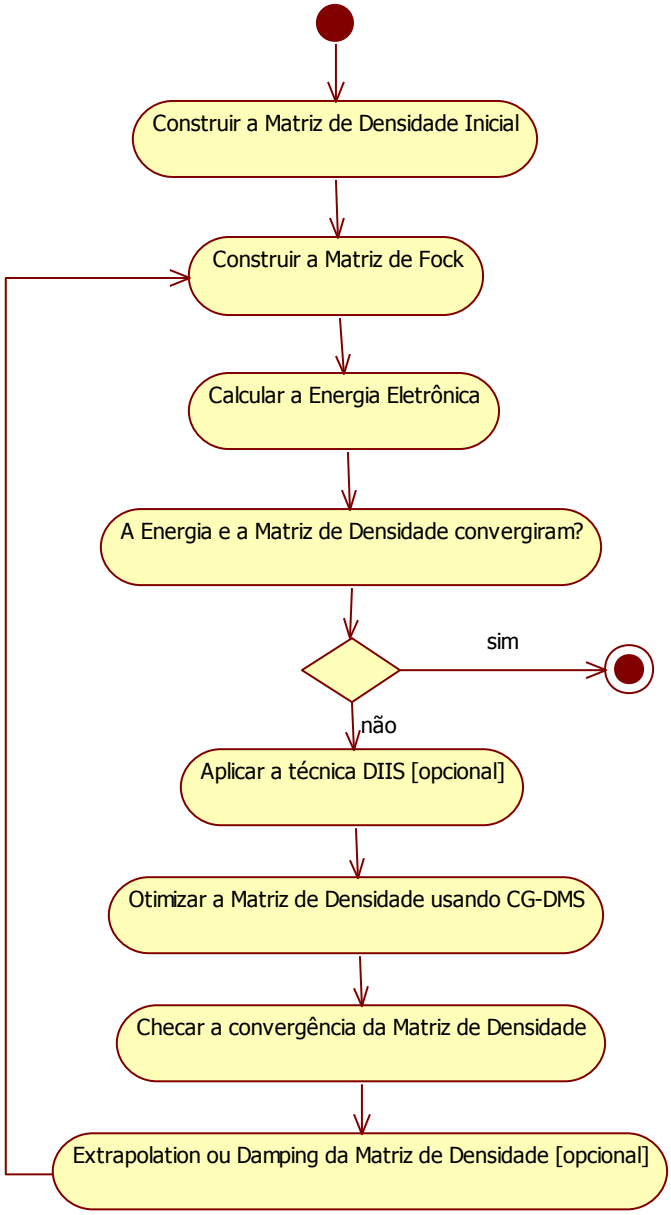

Figura 4: Diagrama de atividades do método SCF com a técnica CG-DMS

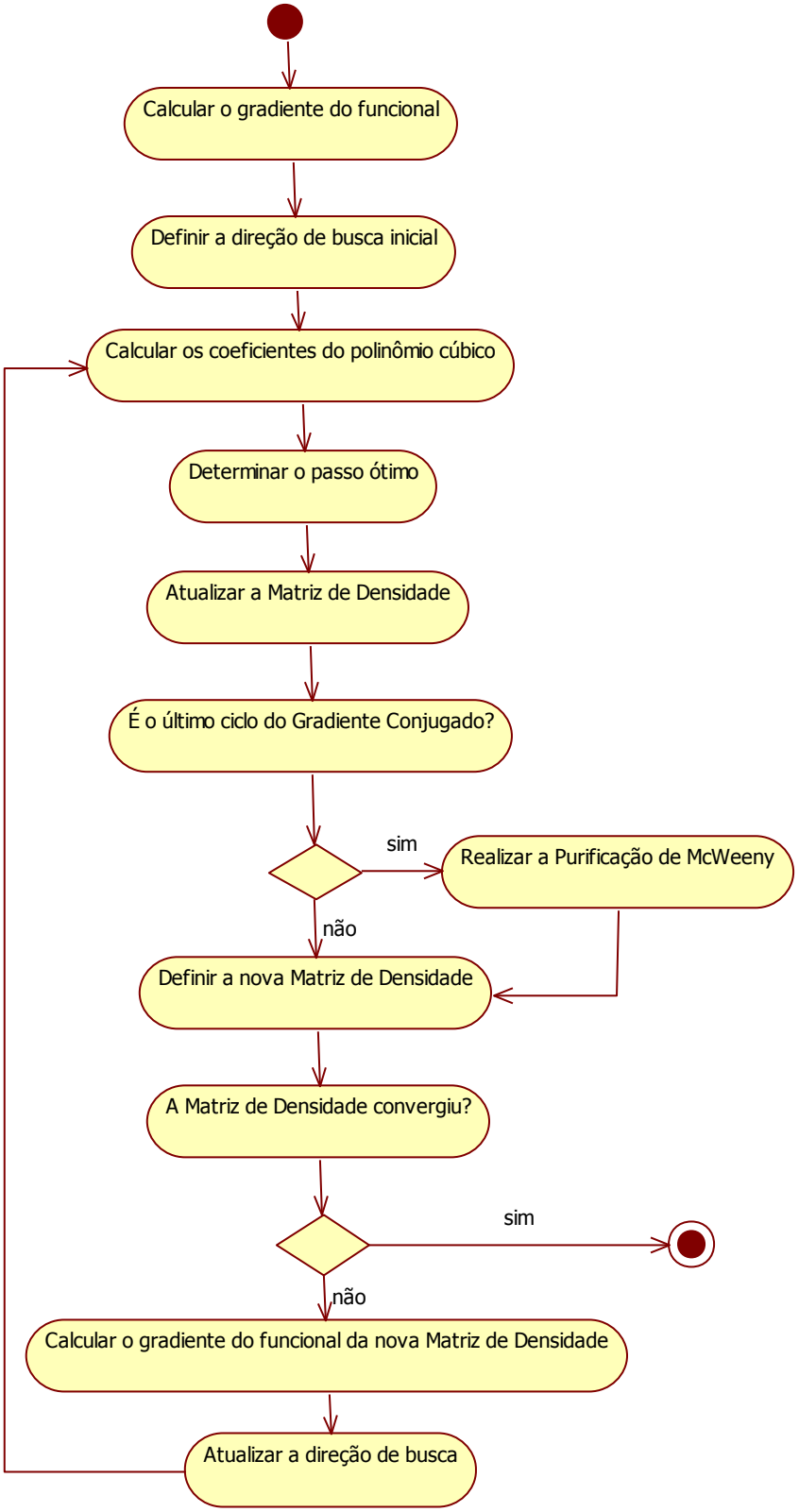

Figura 5: Diagrama de atividades da técnica CG-DMS

2.2.4 Métodos de Purificação

A purificação da matriz de densidade especifica um modo de encontrar a matriz de densidade sem o conhecimento explícito dos orbitais, evitando a diagonalização da matriz de Fock. Sabe-se também que a matriz de densidade possui as seguintes propriedades:

- Qualquer autovetor da matriz de Fock também é autovetor da matriz de densidade;
- Os autovalores da matriz de densidade ou são 0 ou são 1. Portanto, a matriz de densidade é idempotente;
- Os n_{occ} autovetores da matriz de Fock que correspondem aos menores autovalores são autovetores da matriz de densidade com valor 1, enquanto que, os outros autovetores da matriz de densidade tem autovalor 0.

Ou seja, a partir de uma matriz de Fock pode-se encontrar a matriz de densidade movendo os n_{occ} menores autovalores para 1 e os autovalores restantes para 0.

Os métodos de purificação primeiro se basearam no polinômio de McWeeny mostrado na Equação 17. De acordo com McWeeny, a matriz de densidade alcançaria a idempotência através do refinamento deste polinômio. Todos os métodos de purificação iniciam a partir de uma matriz Hamiltoniana efetiva escalonada e deslocada. Polinômios de baixa ordem são, então, recursivamente aplicados para construir uma expansão da *step function* que é mostrada na Equação 10. Na aritmética exata, o subespaço ocupado da matriz Hamiltoniana efetiva é preservada através deste procedimento, já que o polinômio influencia somente os autovalores. Nos cálculos práticos, contudo, erros ocorrem a cada iteração e ao contrário dos métodos de minimização, qualquer precisão perdida não pode ser recuperada.

Entre os esquemas de Purificação, aqueles que ganharam destaque na comunidade científica foram:

- *Grand Canonical* (PALSER e MANOLOPOULOS, 1998) – usa um potencial químico pré-definido;
- *Canonical Trace Conserving* (PALSER e MANOLOPOULOS, 1998) – usa um valor de traço fixo em cada passo de iteração;

- *Trace Correcting* (NIKLASSON, 2002) – o valor do traço correto somente é alcançado na convergência;
- *Trace Re-Setting* (NIKLASSON, TYMCZAK e CHALLACOMBE, 2003) – o valor do traço não é rigorosamente conservado, necessitando de um mecanismo de *re-setting* se o domínio da convergência é excedido.

Nas seções posteriores serão apresentadas as ideias fundamentais destes métodos de purificação.

2.2.5 Grand Canonical Purification

Neste esquema de purificação a matriz de densidade inicial é calculada pela Equação 24:

$$\rho_0 = \frac{\lambda}{2}(\mu I - H) + \frac{1}{2} I. \tag{24}$$

Onde o valor de λ é dado pela Equação 25:

$$\lambda = min\left\{\frac{1}{H_{max} - \mu}, \frac{1}{\mu - H_{min}}\right\}. \tag{25}$$

Os valores posteriores da matriz densidade são calculados de acordo com a Equação 26:

$$\rho_{n+1} = 3\rho_n^2 - 2\rho_n^3. \tag{26}$$

Conforme é apresentado em (PALSER e MANOLOPOULOS, 1998), este algoritmo requer menos esforço que os algoritmos de minimização, tanto porque requer menos operações de multiplicação de matrizes, quanto porque converge quadraticamente. Porém requerem informações adicionais, pois são necessários os valores H_{max} e H_{min}, respectivamente, maior e menor autovalor da matriz Hamiltoniana. Estes autovalores podem

ser facilmente estimados usando as fórmulas de Gershgorin que são apresentadas nas Equações 27 e 28:

$$H_{min} = \min_{i} \left\{ H_{ii} - \sum_{j \neq i} |H_{ij}| \right\}, \tag{27}$$

$$H_{max} = \max_{i} \left\{ H_{ii} + \sum_{j \neq i} |H_{ij}| \right\}. \tag{28}$$

2.2.6 Canonical Purification

Este é outro método de Purificação cuja característica principal é estabelecer um número fixo de elétrons N_{el} e um potencial químico μ definido para a realização dos cálculos. O chute inicial ótimo a partir destas condições é dado pela Equação 29:

$$\rho_0 = \frac{\lambda}{N}(\bar{\mu} I - H) + \frac{N_{el}}{N} I. \tag{29}$$

Os valores de $\bar{\mu}$ e λ são calculados a partir das Equações 30 e 31:

$$\bar{\mu} = \frac{tr[H]}{N}, \tag{30}$$

$$\lambda = min\left\{ \frac{N_{el}}{H_{max} - \bar{\mu}}, \frac{N - N_{el}}{\bar{\mu} - H_{min}} \right\}. \tag{31}$$

Os valores seguintes da matriz de densidade são calculados a partir da função expressa na Equação 32:

$$\rho_{n+1} = \begin{cases} \dfrac{[(1-2c_n)\rho_n + (1+c_n)\rho_n^2 - \rho_n^3]}{(1-c_n)}, & se\ c_n \leq \dfrac{1}{2} \\ \dfrac{[(1+c_n)\rho_n^2 - \rho_n^3]}{c_n}, & se\ c_n \geq \dfrac{1}{2} \end{cases} \quad (32)$$

O valor de c_n pode ser calculado a partir da Equação 33:

$$c_n = \frac{tr[\rho_n^2 - \rho_n^3]}{tr[\rho_n - \rho_n^2]}. \quad (33)$$

2.2.7 Trace Re-Setting Density Matrix Purification

Esta técnica emprega um polinômio de purificação com propriedades de convergência ótimas F, aliado a um polinômio de re-ocupação G, tal que a combinação linear mostrada na Equação 34 produza o traço correto da Equação 35 quando γ_n está no intervalo de aplicabilidade, ou seja, $\gamma_n \in [\gamma_{min}, \gamma_{max}]$.

$$X_{n+1} = F(X_n) + \gamma_n G(X_n), \quad (34)$$

$$Tr[X_{n+1}] = N_{el}, \quad (35)$$

$$\gamma_n = \frac{(N_{el} - Tr[F(X_n)])}{Tr[G(X_n)]}. \quad (36)$$

Quando o valor de γ_n está fora do intervalo, um mecanismo auxiliar é usado para redefinir a solução, trazendo o traço de volta para o regime de conservação, tal qual é apresentado na função representada pela Equação 37:

$$P(x) = \begin{cases} x^2 & \gamma < \gamma_{min} \\ 2x - x^2 & \gamma > \gamma_{max} \end{cases} \quad (37)$$

Em (NIKLASSON, TYMCZAK e CHALLACOMBE, 2003) é mostrado que os polinômios F e G que representam uma melhor escolha são aqueles apresentados nas Equações 38 e 39:

$$F(x) = x^2(4x - 3x^2), \tag{38}$$

$$G(x) = x^2(1 - x)^2. \tag{39}$$

2.2.8 Matrizes Esparsas

Todos os métodos da matriz de densidade de escalonamento linear utilizam o conceito de matrizes esparsas. Uma das definições de esparsidade de uma matriz se refere à porcentagem de elementos desprezíveis de uma matriz em relação ao número total de elementos desta matriz. Sendo assim, matrizes com um grande número de elementos desprezíveis ou próximos de zero são consideradas matrizes esparsas. Em estudos realizados no campo da Química Quântica (MASLEN, OCHSENFELD, *et al.*, 1998), observou-se que era comum surgir matrizes esparsas nos cálculos e o modo como elas são tratadas é um fator crucial para o desempenho da aplicação.

Para o surgimento de matrizes esparsas é necessário remover elementos insignificantes das matrizes e esta operação é conhecida como truncamento. Existem basicamente duas abordagens de truncamento. Na primeira abordagem, elementos das matrizes cuja magnitude seja menor do que um limiar pré-definido serão anulados. Na segunda abordagem, elementos insignificantes das matrizes são anulados quando a distância entre os centros das funções de base são maiores do que um raio de corte pré-definido.

Quando os dados da aplicação estão organizados em matrizes, é necessário escolher qual formato que melhor se encaixa nas necessidades da aplicação e quais estruturas de dados serão utilizadas. Alguns dos critérios que devem ser observados para o tratamento de matrizes são:

- Utilização de memória;
- Velocidade das operações:
    - Sobrecarga;
    - Quantidade de operações de ponto flutuante;
- Acessibilidade de elementos individuais da matriz;
- Flexibilidade.

É comum utilizar o formato denso (matriz na forma completa ou padrão) para representar matrizes de dados devido a sua facilidade de implementação. Porém este formato não é apropriado para tratar as matrizes esparsas, pois vai sempre armazenar elementos desprezíveis e realizar operações de ponto-flutuante desnecessárias sobre estes elementos. Detalhes como a realização de operações desnecessárias sobre certos elementos da matriz limitam o desempenho alcançado pelos métodos numéricos de álgebra linear. Novos métodos de álgebra linear devem ser desenvolvidos para melhorar o desempenho destas operações quando aplicadas a matrizes esparsas.

Para facilitar a manipulação de matrizes esparsas foram criados diversos formatos com características bem específicas. Os formatos mais usados pela comunidade são:

- Banded Linpack format (BND);
- Compressed Sparse Row format (CSR);
- Compressed Sparse Column (CSC);
- Coordinate format (COO).

Pela estrutura das matrizes geradas nos cálculos de química quântica, têm-se dado mais relevância aos formatos CSR e CSC e também a uma variação destes que é o BCSR (*Blocked Compressed Sparse Row*) conforme (RUBENSSON, 2005). Os formatos *Compressed Sparse Row* (CSR) e *Compressed Sparse Column* têm como objetivo principal utilizar a menor quantidade de memória. A diferença entre estes dois formatos é que no formato CSR são utilizados apontadores para as linhas, enquanto que, no formato CSC são utilizados apontadores para as colunas. São estas as vantagens destes formatos:

- Pouca utilização de memória;
- Operações com números de ponto flutuante reduzido.

No entanto, estes formatos são pobres na flexibilidade, como, por exemplo, se inserir um novo elemento na matriz, e lentos ao acessar elementos individuais das matrizes.

Para sua representação são usados três vetores para guardar uma matriz esparsa no formato CSR. São eles:

- Um vetor denominado *A* de números reais, com tamanho igual ao número de elementos diferentes de zero na matriz;
- Um vetor denominado *JA* de números inteiros, com o mesmo tamanho de *A*. Este vetor armazena os índices das colunas dos valores de *A*.
- Um vetor denominado *IA* de números inteiros, com tamanho igual ao número de linhas da matriz mais um. Os valores de *IA* armazenam ponteiros para o início de cada linha em *A* e em *JA*. O último elemento do vetor indica onde a linha n+1 iniciaria.

A Figura 6 traz um exemplo de uma matriz esparsa:

$$A = \begin{bmatrix} 10 & 0 & 0 & 0 & -2 & 0 \\ 3 & 9 & 0 & 0 & 0 & 3 \\ 0 & 7 & 8 & 7 & 0 & 0 \\ 3 & 0 & 8 & 7 & 5 & 0 \\ 0 & 8 & 0 & 9 & 9 & 13 \\ 0 & 4 & 0 & 0 & 2 & -1 \end{bmatrix}.$$

Figura 6: Matriz Esparsa

A representação da matriz da Figura 6 no formato CSR ficará da seguinte forma, considerando o zero como índice base:

A: [10, -2, 3, 9, 3, 7, 8, 7, 3, 8, 7, 5, 8, 9, 9, 13, 4, 2, -1]
IA: [0, 2, 5, 8, 12, 16, 19]
JA: [0, 4, 0, 1, 5, 1, 2, 3, 0, 2, 3, 4, 1, 3, 4, 5, 1, 4, 5]

No formato CSC também são utilizados três vetores sendo que a diferença está nos vetores *IA* e *JA*, pois o vetor *IA* armazena os índices das linhas e o vetor *JA* armazena

ponteiros para o início de cada coluna. A representação CSC corresponde a transposta da representação CSR.

Uma biblioteca específica para o tratamento de matrizes esparsas é a SPARSKIT2 (SAAD, 1990). A vantagem de se utilizar a SPARSKIT2 é que a maioria de suas operações utiliza como base os formatos CSR e CSC que permitem uma menor utilização do espaço de memória.

Essa biblioteca apresenta uma versão da biblioteca de álgebra linear BLAS modificada para o tratamento de matrizes bastante esparsas chamada de BLASSM. Essa parte da SPARSKIT2 contém diversas operações necessárias tais como: adição e multiplicação de matrizes esparsas. Porém, o desempenho numérico desta biblioteca para matrizes esparsas ainda não é tão satisfatório quanto o seu desempenho para matrizes no formato denso. Isto porque matrizes esparsas não possuem uma estrutura que possa ser definida antes da realização dos cálculos, dificultando a escrita de códigos otimizados para operações de álgebra linear com matrizes esparsas.

Outras bibliotecas que contém operações de álgebra linear básicas para matrizes esparsas são a Sparse BLAS da MKL (Math Kernel Library from Intel - Intel Software Network) e a PSBLAS (FILIPPONE e COLAJANNI, 2000).

Uma biblioteca para tratamento de matrizes esparsas de forma paralela é a CUSPARSE (NAUMOV, CHIEN, *et al.*, 2010). Esta biblioteca contém um conjunto de operações de álgebra linear usadas para tratamento de matrizes esparsas e projetadas para ser chamada de um código C/C++. Outro fato relevante é que esta biblioteca foi escrita no modelo de programação CUDA, que será discutido mais adiante, e utiliza recursos computacionais provenientes das GPUs da NVIDIA. As operações desta biblioteca estão divididas em quatro categorias:

- Nível 1: inclui operações entre um vetor no formato esparso e um vetor no formato denso;
- Nível 2: inclui operações entre uma matriz no formato esparso e um vetor no formato denso;
- Nível 3: inclui operações entre uma matriz no formato esparso e um conjunto de vetores (*tall matrix*) no formato denso;
- Nível 4: inclui operações de conversão que permitem a conversão entre os diferentes formatos de matriz.

Dentre os formatos de matriz usados pela CUSPARSE, está também o formato CSR. Inclusive existe uma operação de multiplicação de matrizes que envolve matriz esparsa no formato CSR que será discutida mais adiante.

Um problema recorrente com as bibliotecas de álgebra linear para matrizes esparsas é a sua dificuldade de implementação, pois a escolha do algoritmo é altamente dependente da estrutura de não-zeros da matriz e da arquitetura da máquina-alvo. Enquanto que o ajuste em relação à arquitetura pode usualmente ser feito com antecedência, a adaptação do algoritmo para um padrão arbitrário de não-zeros deve ser feito em tempo de execução, já que a matriz não é conhecida de antemão.

2.3 MOPAC

O MOPAC é um pacote que permite cálculos semi-empíricos de orbitais moleculares para o tratamento de diversos sistemas moleculares e sólidos. Seu código é escrito em FORTRAN 90 e possui cerca de 30.000 linhas de código. Os métodos semi-empíricos que podem ser usados são: MNDO (DEWAR e THIEL, 1977), AM1 (DEWAR e HEALY, 1985), RM1 (ROCHA, FREIRE, *et al.*, 2006) e PM6 (STEWART, 2007). Esse programa pode ser usado para computar diversas propriedades moleculares, entre elas: geometrias moleculares, espectro vibracional, quantidades termodinâmicas, efeitos da substituição isotópica e constantes de força para moléculas, radicais, íons e polímeros.

O programa MOPAC possui uma estrutura codificada que pode ser visualizada a partir de um diagrama de fluxo na Figura 7.

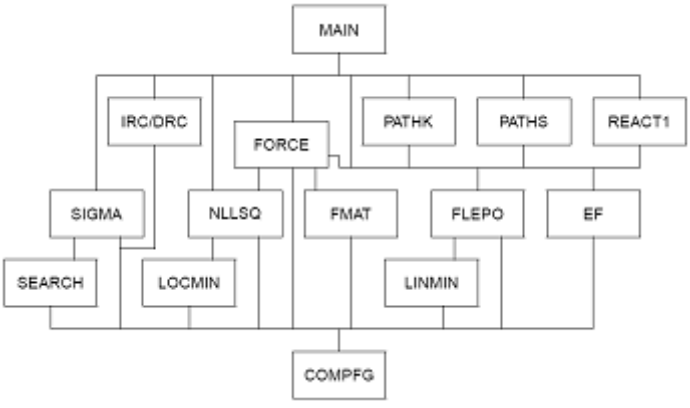

Figura 7: Diagrama de fluxo do MOPAC

Esse diagrama de fluxo mostra como o cálculo de estrutura eletrônica acontece dentro desse programa, nas mais variadas modalidades. Não será detalhado cada procedimento, mas será dado um destaque especial para os procedimentos mais importantes para o trabalho como:

- COMPFG: cálculo SCF de Hartree-Fock-Roothaan e o cálculo das primeiras derivadas da energia com relação às coordenadas nucleares;
- FORCE: cálculo das segundas derivadas da energia com relação às coordenadas nucleares mais o cálculo das freqüências vibracionais e análise de dados termoquímicos;
- FLEPO: otimização de geometria usando o método BFGS (BROYDEN, 1970);
- EF: otimização de geometria usando o método Eigenvector Following;
- IRC/DRC: cálculo de coordenada de reação intrínseco/dinâmico.

Nesses procedimentos podem-se destacar as seguintes tarefas computacionais que são custosas para estes cálculos:

- Cálculo das integrais atômicas, principalmente as de repulsão inter-eletrônica de dois-centros;
- Montagem da matriz de Fock;
- Diagonalização da matriz de Fock;
- Procedimento iterativo SCF;
- Cálculo das primeiras derivadas da energia com respeito às coordenadas atômicas;
- Cálculo das segundas derivadas da energia com respeito às coordenadas atômicas;
- Diagonalização da matriz Hessiana.

Os procedimentos listados fazem parte do procedimento iterativo SCF para o cálculo da energia total do sistema molecular. Como já foi descrito, o procedimento que executa tal operação é o COMPFG, mais especificamente o procedimento ITER, como pode ser visto na Figura 8, que mostra todos os procedimentos chamados pela COMPFG.

No procedimento COMPFG ainda ocorre o cálculo das primeiras derivadas da energia com respeito às coordenadas atômicas. Essas derivadas primeiras são os gradientes da energia e são usadas pelo procedimento que realiza a otimização de geometria, seja através do método BFGS (procedimento FLEPO) ou pelo método EF (procedimento EF). O cálculo dos gradientes numéricos através de diferenças finitas é realizado pelo procedimento DERIV.

A etapa mais custosa num cálculo semi-empírico convencional é a diagonalização da matriz de Fock que é um algoritmo de complexidade $O(n^3)$, onde o parâmetro n pode ser o número de elétrons da molécula. Na Figura 8, esta etapa é executada pelo procedimento DIAG. Os outros procedimentos associados ao SCF tem ordem de complexidade $O(n^2)$. Por esse motivo, a maioria dos esforços é concentrada na tentativa de se reduzir o tempo gasto no procedimento numérico que diagonaliza a matriz de Fock.

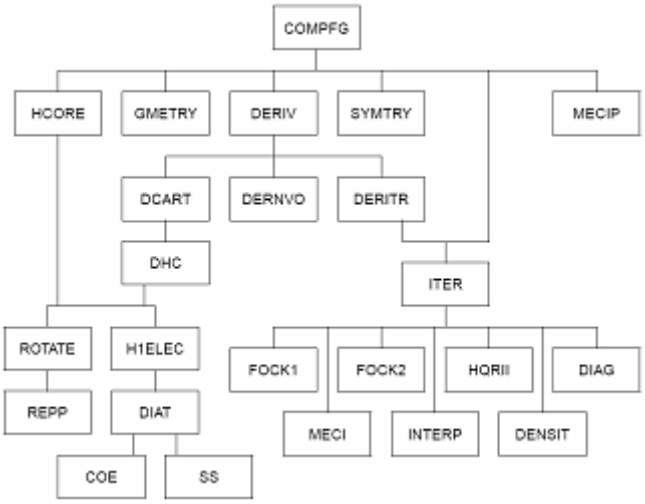

Figura 8: Diagrama de fluxo para um cálculo SCF no MOPAC

2. 5 Computação Paralela de Alto Desempenho

A computação paralela é uma forma de computação na qual o objetivo é realizar tarefas de modo simultâneo, partindo do princípio que estas tarefas podem ser divididas em tarefas menores, possibilitando o paralelismo. Já que o objetivo principal da computação paralela é obter um desempenho melhor, a otimização de uma aplicação paralela é parte integral do processo de desenvolvimento de programas. Para obter o melhor desempenho se faz necessário seguir algumas boas práticas como: conhecer bem a arquitetura onde será

desenvolvida a aplicação; estabelecer uma estratégia de paralelização do código da aplicação; e traçar um mapeamento do código da aplicação e do seu modelo de programação para a arquitetura.

Hoje em dia, grandes processadores ineficientes estão sendo substituídos por vários processadores menores que trabalham cooperativamente para possibilitar um melhor desempenho. A maioria dos problemas científicos necessita de computadores cada vez mais rápidos e por isso a indústria produz *softwares* e *hardwares* que facilitem a escrita correta de programas de processamento paralelo capazes de executarem de forma mais eficiente à medida que o número de *cores* de processamento por *chip* aumente.

Um modo comum de categorizar os *hardwares* se baseia no número de fluxos de instruções e de fluxos de dados que é a classificação de Flynn. Esta classificação pode ser observada na Figura 9:

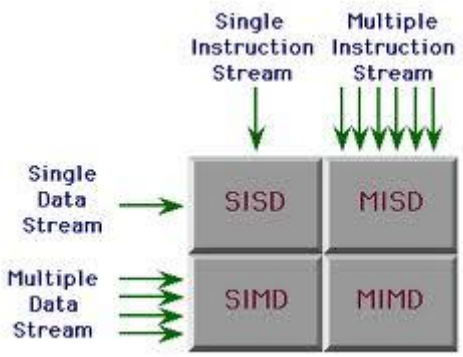

Figura 9: Arquiteturas Flynn

Arquiteturas do tipo SISD (*Single Instruction Single Data*) consistem num único fluxo de instruções operando num único fluxo de dados. Estas arquiteturas possuem processamento seqüencial característico da máquina de Von Neumann e pode ser encontradas em computadores pessoais e em estações de trabalho. As instruções, porém, podem ser executadas de forma sobreposta em diferentes estágios, o que caracteriza a técnica *pipeline*. Possuem apenas uma única unidade de controle e uma ou mais unidades funcionais. A Figura 10 mostra as características desta arquitetura:

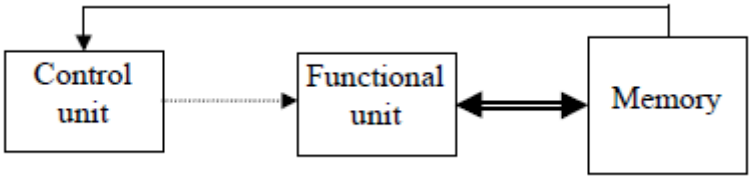

Figura 10: SISD

A arquitetura SIMD (*Single Instruction Multiple Data*) se caracteriza pelo processamento de vários dados obedecendo a uma única instrução também de forma sequencial. Também possui uma única unidade de controle e várias unidades funcionais que operam sobre diferentes módulos de memória. Esta arquitetura pode ser encontrada em processadores vetoriais e matriciais. Os detalhes desta arquitetura podem ser observados na Figura 11:

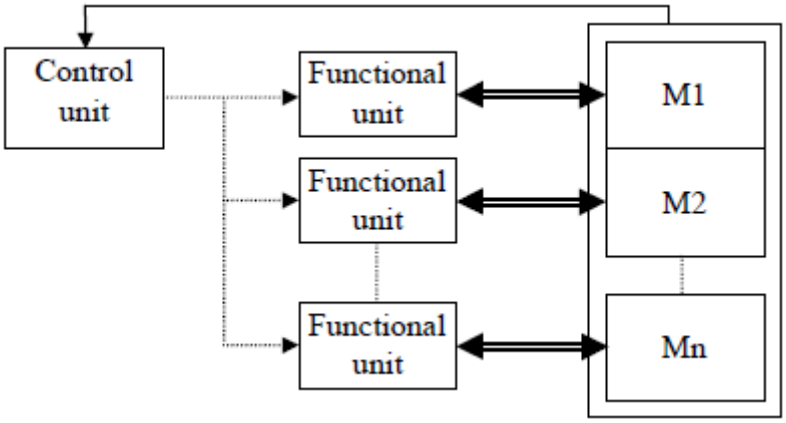

Figura 11: SIMD

Na arquitetura MISD (*Multiple Instruction Single Data*) tem-se múltiplas unidades de controle distintas que operam sobre o mesmo dado. Não existem exemplos de tal arquitetura hoje em dia. A Figura 12 representa esta arquitetura:

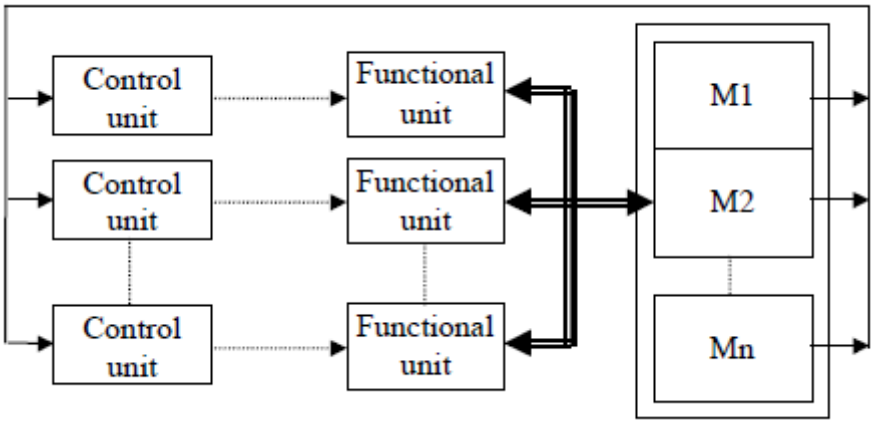

Figura 12: MISD

Na arquitetura MIMD (*Multiple Instruction Multiple Data*) são processados múltiplos dados por múltiplas instruções. Cada unidade de controle comanda sua unidade funcional que vai operar seu módulo de memória. Os multiprocessadores formam um exemplo desta arquitetura. A Figura 13 ilustra esta arquitetura:

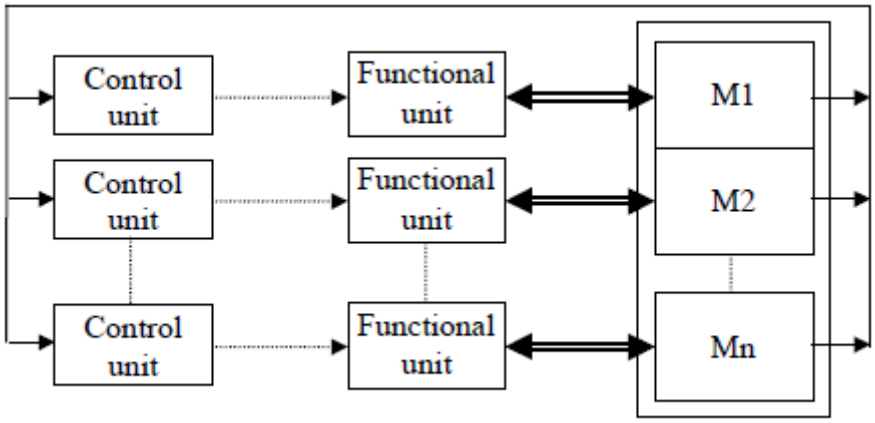

Figura 13: MIMD

Em relação ao tipo de memória, as arquiteturas paralelas podem possuir memória compartilhada ou distribuída. Na arquitetura de memória compartilhada, todos os dados acessados pela aplicação ocupam uma memória global que é acessível por todos os processadores paralelos. Isto significa que cada processador pode buscar e armazenar dados em qualquer lugar na memória independentemente. Este modelo de programação é caracterizado pela necessidade de sincronização para preservar a integridade das estruturas de dados compartilhados.

Na arquitetura de memória distribuída, os dados são vistos como sendo associados aos processadores em particular, então a comunicação é requerida para acessar locais de dados remotos via *Message-Passing*. Geralmente, para pegar os dados de uma memória remota, o processador proprietário do dado deve enviá-lo e o processador requisitante deve recebê-lo. Neste modelo, as primitivas de *send* e *receive* tomam o lugar da sincronização.

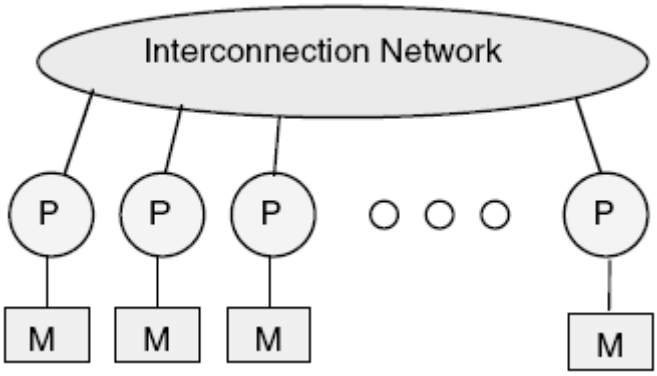

Figura 14: Arquitetura de Memória Distribuída

O objetivo da computação paralela é ter o tempo de execução de uma aplicação reduzido por um fator que é inversamente proporcional ao número de processadores usados. Um modo de se dizer que um programa é escalável é através do *speedup*, que é definido como a taxa do tempo de execução de um programa num único processador pelo tempo de execução na configuração paralela como pode ser visto na Equação 40:

$$Speedup(n) = \frac{T(1)}{T(n)}. \tag{40}$$

O *speedup* também pode ser expresso através da Lei de Amdahl que é usada para encontrar a melhoria máxima esperada de um sistema quando somente uma parte do sistema é melhorada. A Lei de Amdahl pode ser calculada de acordo com a Equação 41:

$$S = \frac{1}{(1-P) + \frac{P}{N}}. \tag{41}$$

Uma aplicação é dita escalável se o *speedup* com *n* processadores está próximo do valor *n*. Há três razões principais para que a escalabilidade não seja alcançada em algumas

aplicações. Primeiro, a aplicação pode ter uma grande região que deve ser executada de forma seqüencial. Um segundo impedimento para a escabilidade é a necessidade de um alto grau de comunicação ou coordenação. O terceiro maior impedimento para a escalabilidade é o pobre balanceamento de carga.

2.6 CPU *X* GPU

A frequência de *clock* de um *core* na CPU não tem aumentado ultimamente devido às limitações físicas como a dissipação de calor. Uma alternativa a este problema são os processadores *multicore* que são compostos de dois ou mais *cores* de processamento em um mesmo *chip* de CPU. Porém esta alternativa aumenta a complexidade para os programadores fazendo com que se torne mais difícil para as aplicações explorarem esta tecnologia.

As GPUs se apresentam como uma alternativa para programação massiva paralela de dados, pois sua arquitetura foi desenhada para garantir altas taxas de processamento de vídeos impulsionadas pela indústria de jogos. Elas realizam um grande número de cálculos de ponto-flutuante por *frame* do vídeo e apresentam uma grande quantidade de *threads* de execução que são organizadas em *grids* computacionais. A Figura 15 e a Figura 16 mostra uma evolução comparativa entre os *hardwares* existentes:

As diferenças das arquiteturas da CPU e da GPU são responsáveis por esta diferença no desempenho. A arquitetura da CPU é otimizada para obter proveito de códigos seqüenciais, através da utilização de controladores lógicos sofisticados que permitem às instruções das *threads* de execução serem executadas em paralelo ou até mesmo fora de sua ordem seqüencial mantendo ainda a aparência de uma execução seqüencial.

Além disso, as CPUs ainda possuem as memórias *cache* que estão cada vez maiores para possibilitarem a redução de latência de instruções e de acessos a dados por aplicações extremamente complexas.

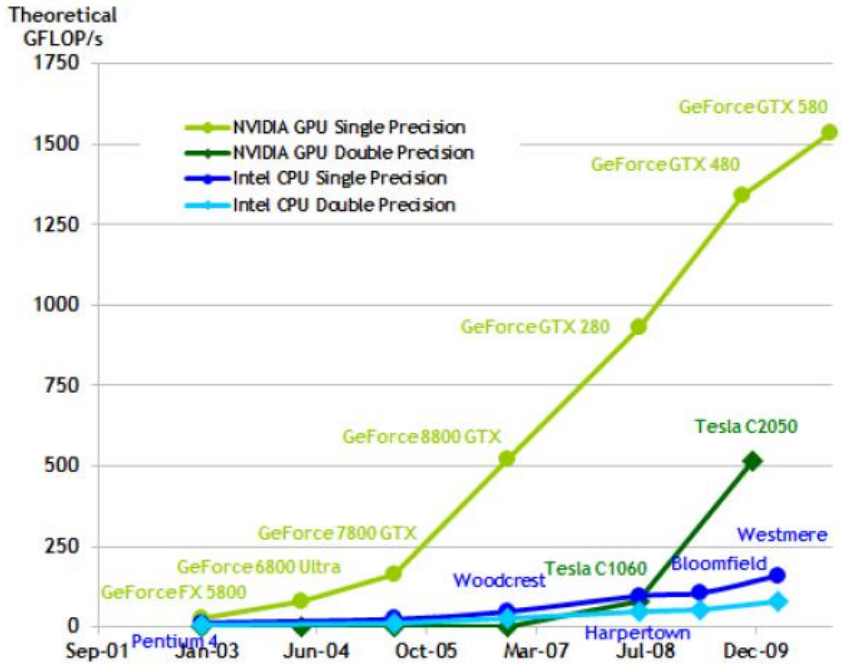

Figura 15: GFLOP/s em algumas CPUs e GPUs

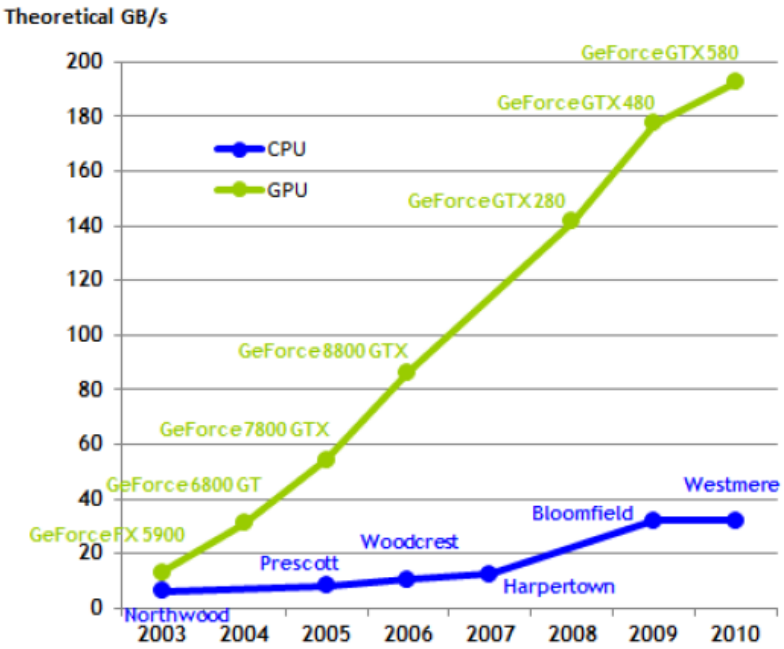

Figura 16: GB/s em algumas CPUs e GPUs

Outra diferença está na *memory bandwidth*, pois as GPUs estão operando a aproximadamente 10 vezes mais que o *bandwidth* das atuais CPUs. Devido a questões como o *frame buffer* e do modelo de *relaxed memory* – o modo como vários sistemas, aplicações, dispositivos de E/S esperam que os acessos à memória ocorram – a *memory bandwidth* das

CPUs não estão melhorando muito. Enquanto isso, as arquiteturas das GPUs possuem modelos de memória mais simples o que facilita o aumento da *memory bandwidth*. Pequenas memórias *cache* são utilizadas nas GPUs para ajudar a controlar requisitos de *bandwidth* de aplicações possibilitando que múltiplas *threads* acessem os mesmos dados de memória não necessitando acessar a memória toda vez. Como resultado, uma área maior do *chip* da GPU é dedicada para cálculos de ponto-flutuante. A Figura 17 mostra esta diferença arquitetural:

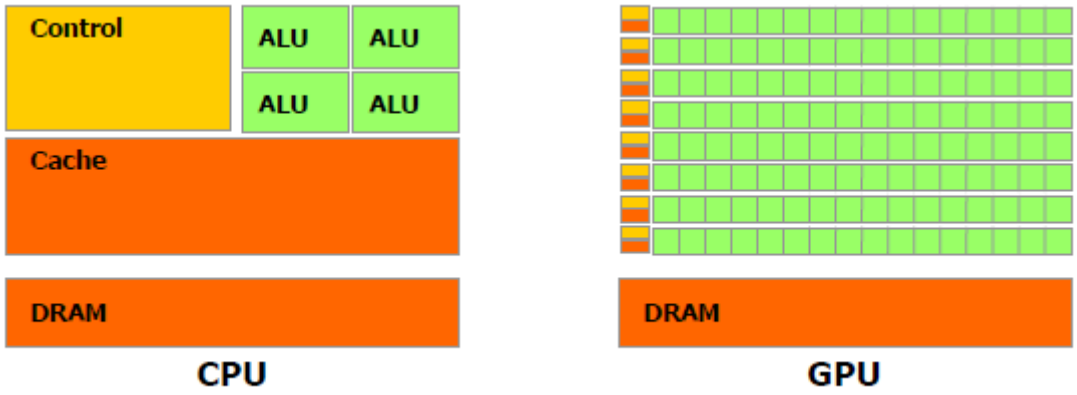

Figura 17: Diferenças nas arquiteturas da CPU e GPU

2.7 Modelo de programação CUDA

CUDA é um modelo de programação massivamente paralela de alto desempenho que utiliza o poder de processamento das GPUs da NVIDIA. Este modelo foi introduzido formalmente em fevereiro de 2007 e vem conquistando diversos usuários dos campos científicos, da biomedicina, da computação, da análise de risco e da engenharia, devido às características das aplicações nesses campos, as quais são altamente paralelizáveis. Essa tecnologia tem chamado a atenção da comunidade acadêmica devido ao seu grande potencial computacional.

Os programas escritos em CUDA possibilitam:

- A programação para sistemas heterogêneos, isto é CPU + GPU, cada dispositivo possuindo sua própria memória dedicada;

- A escalabilidade de programas em centenas de *cores* e milhares de *threads* das GPUs;
- Que os programadores foquem em programas paralelos, pois é necessário apenas acrescentar extensões mínimas de C/C++.

A mais nova arquitetura para GPU é chamada de Fermi. A arquitetura Fermi trouxe as seguintes melhorias para o *hardware* da GPU:

- Melhoria no desempenho das operações de ponto flutuante com dupla precisão;
- Suporte a *Error Correcting Code (ECC)* para a proteção e correção dos dados na GPU;
- Uma hierarquia de memória *cache* para garantir mais rapidez no uso de memória;
- Aumento da memória compartilhada para acelerar as aplicações;
- Troca de contexto mais rápido entre as aplicações e os gráficos;
- Operações atômicas mais rápidas.

A arquitetura Fermi contém um máximo de 512 CUDA *cores* que são organizados em 16 *StreamingMultiprocessor (SM)* com 32 *cores* cada como é visto na Figura 18. O CUDA *core* é composto de uma unidade de lógica aritmética (*Aritmetic logic unit – ALU*) e uma unidade de ponto flutuante (*Floating point unit – FPU*) conforme a Figura 19.

Esta arquitetura utiliza o novo padrão IEEE 754-2008 para ponto flutuante que provê uma instrução de *fused multiply-add (FMA)* tanto para precisão simples como para dupla precisão. Esta instrução melhora a operação de multiplicação-adição realizando a multiplicação e adição e finalizando com um passo de arredondamento sem perda de precisão na adição.

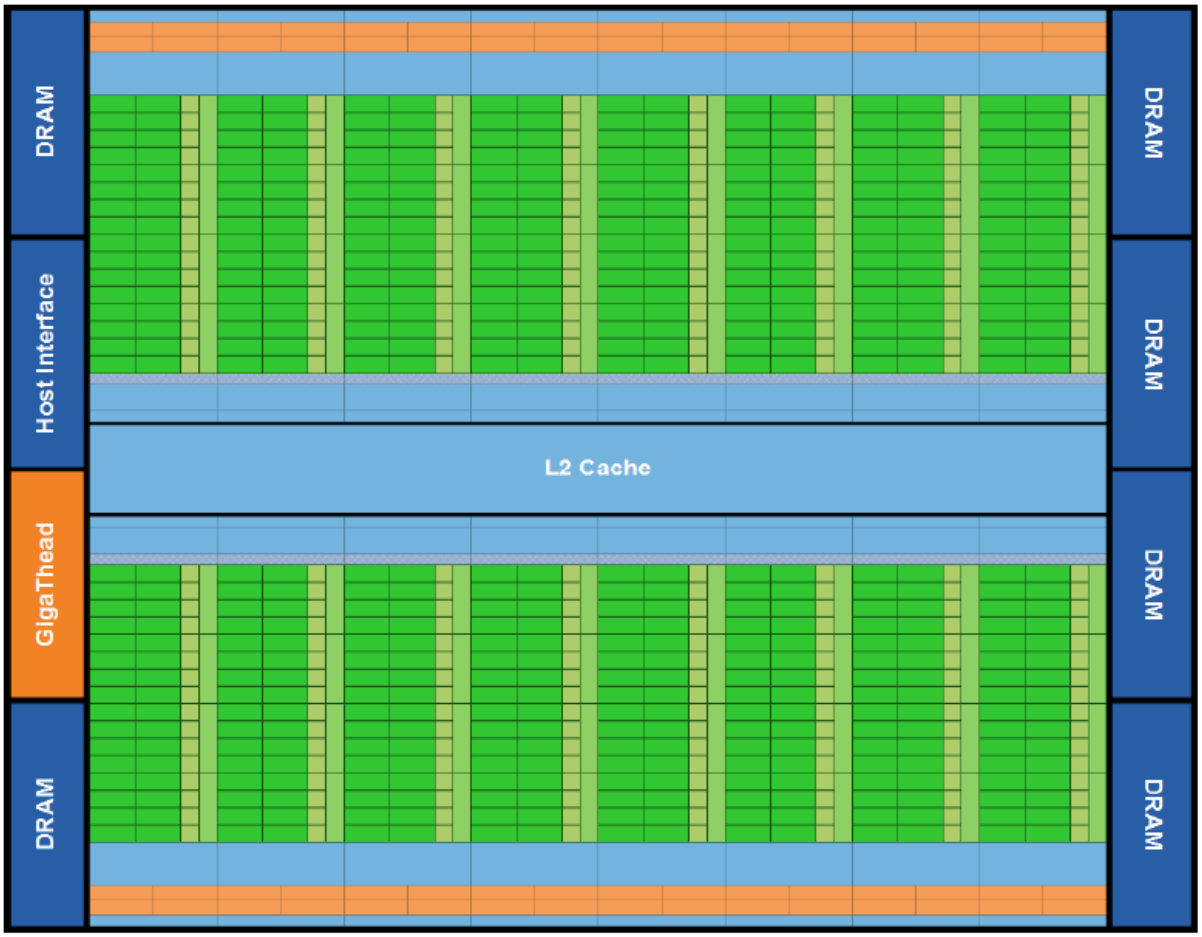

Figura 18: Arquitetura Fermi

Cada SM possui 16 unidades de *load/store*, permitindo que os endereços fonte e destino sejam calculados para 16 *threads* por *clock*. Além disso, cada SM contém quatro unidades de funções especiais (*Special function units – SFU*) que executam instruções transcendentais como: seno, cosseno e raiz quadrada. O SM organiza seus grupos de *threads* em grupos de 32 *threads* paralelas que são chamados *warps*. Cada SM possui dois *warp schedulers* e duas *instruction dispatch units*, permitindo que os dois *warps* sejam executados concorrentemente. Quando as *threads* de um *warp* tomam caminhos divergentes, o *warp* executa sequencialmente ambos os caminhos do código com algumas *threads* inativas, o que faz as *threads* ativas executarem de forma mais lenta. A Figura 20 mostra o *warp scheduler*:

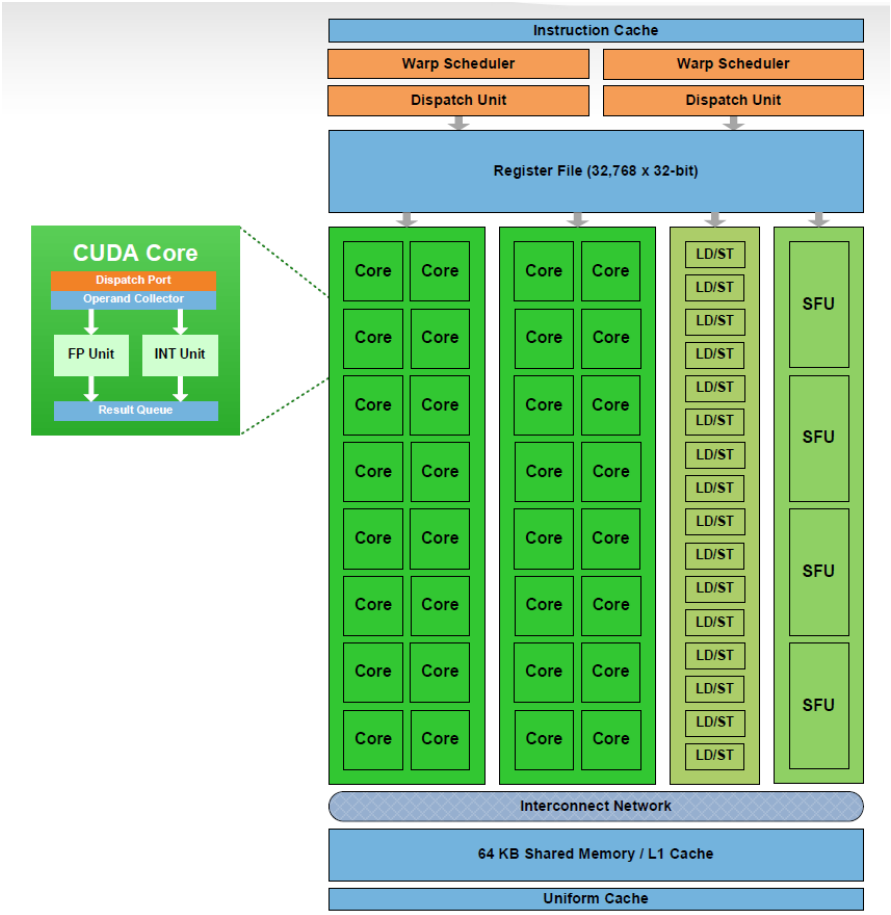

Figura 19: ***Streaming Multiprocessor*** **e CUDA** ***cores***

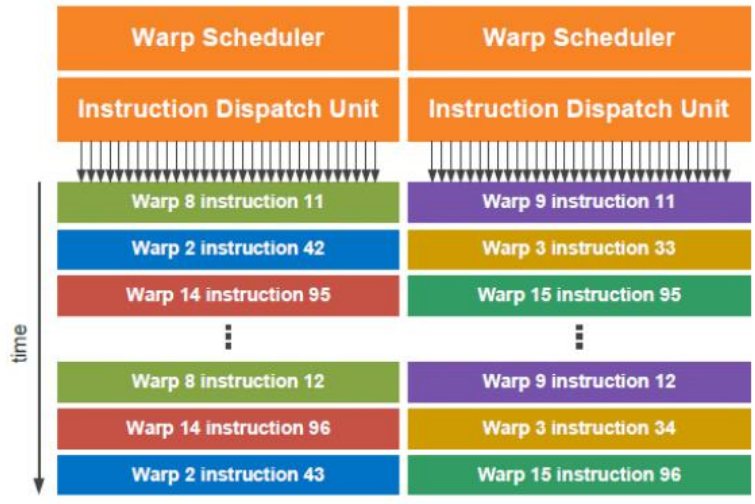

Figura 20: ***Warp Schedulers***

A arquitetura Fermi possui um espaço de endereçamento unificado que unifica estes três espaços de endereçamento para as operações de *load* e *store*: o endereçamento local e

privado das *threads*, o endereçamento da memória compartilhada do bloco e a memória global. Sendo assim, este espaço de endereçamento dá suporte à escrita de programas C++.

Cada GPU atualmente possui memória do tipo GDDR (*Graphics Double Data Rate*) DRAM que constitui a memória global da placa. Este tipo de memória é diferente da memória utilizada pela CPU, pois são memórias utilizadas em aplicações gráficas como memórias de *frame buffer*. Por serem usadas em processos de *render* 3D possuem uma maior *bandwidth*, mas em compensação possuem uma latência maior. Além da memória global, existem a memória compartilhada e dois níveis de memória *cache*. A Figura 21 mostra esta hierarquia:

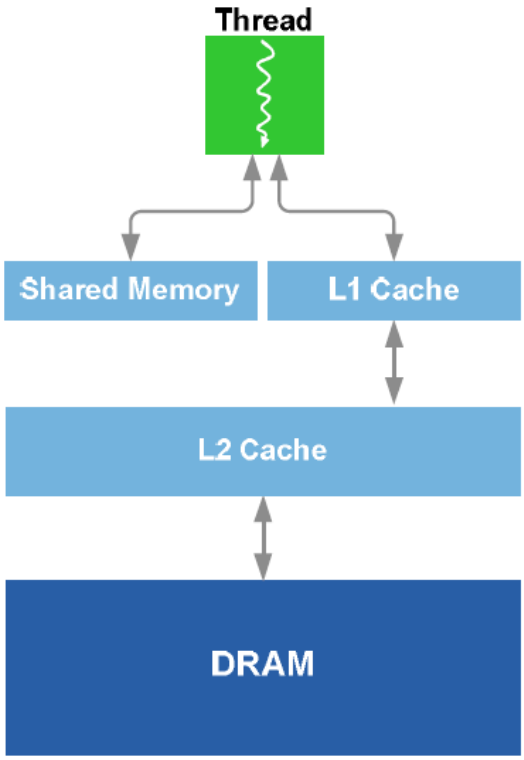

Figura 21: Hieraquia de memória na arquitetura Fermi

Neste universo CUDA surgem dois conceitos importantes: GPU *computing* e GPGPU (*General Purpose computation on GPU)*. GPU *computing* é o termo criado para identificar o uso da GPU para a computação através de uma linguagem de programação paralela e API, sem usar as APIs gráficas tradicionais e o modelo de *pipeline* gráfico. Já GPGPU é uma abordagem que envolve a programação de GPUs para realizar tarefas que não estão relacionadas a gráficos.

O modelo de programação CUDA tem um estilo de *software* SPMD (*single-program multiple data*), na qual o programador escreve um único programa fonte que será decomposto e executado por muitas *threads* em paralelo e em múltiplos processadores da GPU.

No modelo de programação CUDA, a CPU é identificada como o *host* e a GPU é identificada como o *device*. Cada função que é executada no *device* é chamada de *kernel*. CUDA possibilita que um programa serial possua um ou mais *kernels* paralelos e tudo escrito em C/C++, conforme a Figura 22. Os *kernels* tipicamente geram um grande número de *threads* que vão explorar o paralelismo de dados das aplicações.

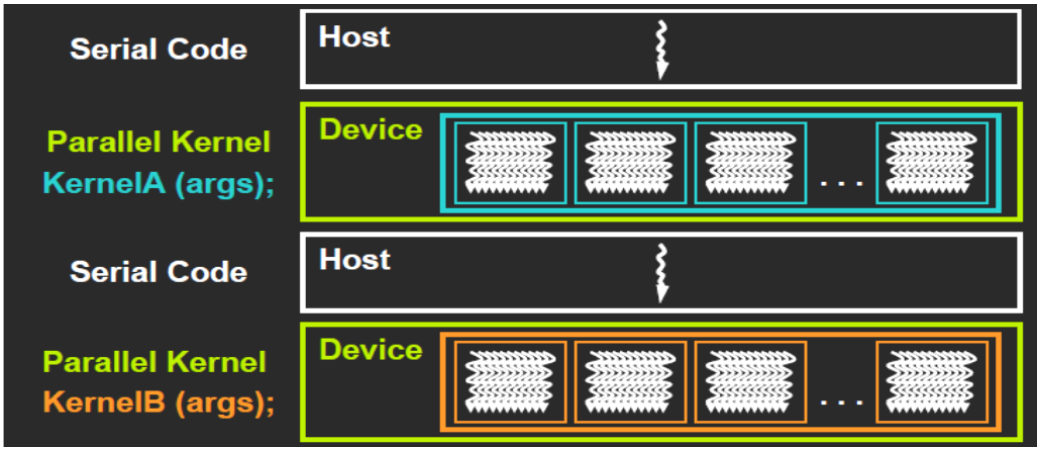

Figura 22: Esquema de um programa escrito em CUDA

As extensões da linguagem C/C++ se classificam em quatro categorias: qualificadores de tipo de função, qualificadores de tipo de variável, diretivas de *kernel* e variáveis *built-in* para indexação de *threads* e blocos.

Os qualificadores de tipo de função são mostrados na Tabela 1 e os qualificadores de tipo de variável são mostrados na Tabela 2:

Tabela 1: Qualificadores de tipo função

Nome do Qualificador	Função
__device__	Especifica que a função é chamada e executada no *device*
__global__	Especifica que a função é um *kernel* a ser chamado pelo *host* e executado pelo *device*
__host__	Especifica que a função é chamada e executada pelo *host*

Tabela 2: Qualificadores de tipo de variável

Nome do Qualificador	Função
__device__	Especifica que a variável reside na memória do *device*. Quando não é especificado nenhum qualificador a variável residirá na memória global do *device*.
__constant__	Especifica que a variável reside na memória constante do *device*
__shared__	Especifica que a variável reside na memória compartilhada do *device*

Os *kernels* de CUDA são executados em paralelo especificando-se um *grid* de blocos, onde cada bloco vai possuir *threads*. Para declarar uma função como sendo um *kernel* utiliza-se o qualificador *__global__*. Para executar um *kernel* e especificar o número de blocos e de *threads* a serem criados, a sintaxe geral utilizada tem a seguinte forma:

Nome_kernel <<<número_de_blocos, número_de_threads >>>(argumentos)

A Figura 23 mostra como as *threads* são organizadas dentro de cada bloco e como cada bloco é organizado no *grid:*

A dimensão do *grid* é especificada pela variável *gridDim* e a dimensão dos blocos é especificada pela variável *blockDim.* Ambas estas variáveis são do tipo *dim3* que possui três componentes (x, y, z). As *threads* podem ser acessadas pela variável *threadIdx,* que possui três componentes, e os blocos podem ser acessados pela variável *blockIdx,* que possui duas componentes.

Existem cinco tipos diferentes de memória no *device*: a memória global, a memória compartilhada, a memória local, a memória de constantes e a memória de textura. Cada um destes tipos de memória tem suas características. A Figura 24 mostra esta hierarquia de memória:

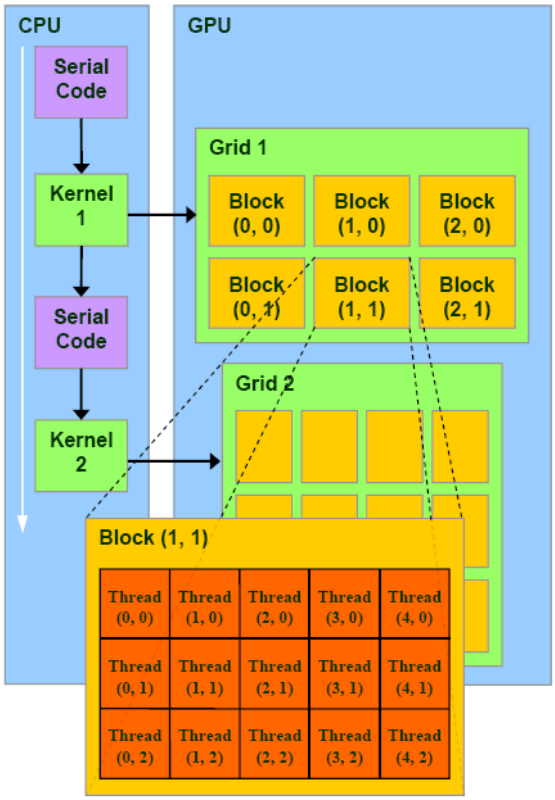

Figura 23: Hierarquia de threads em CUDA

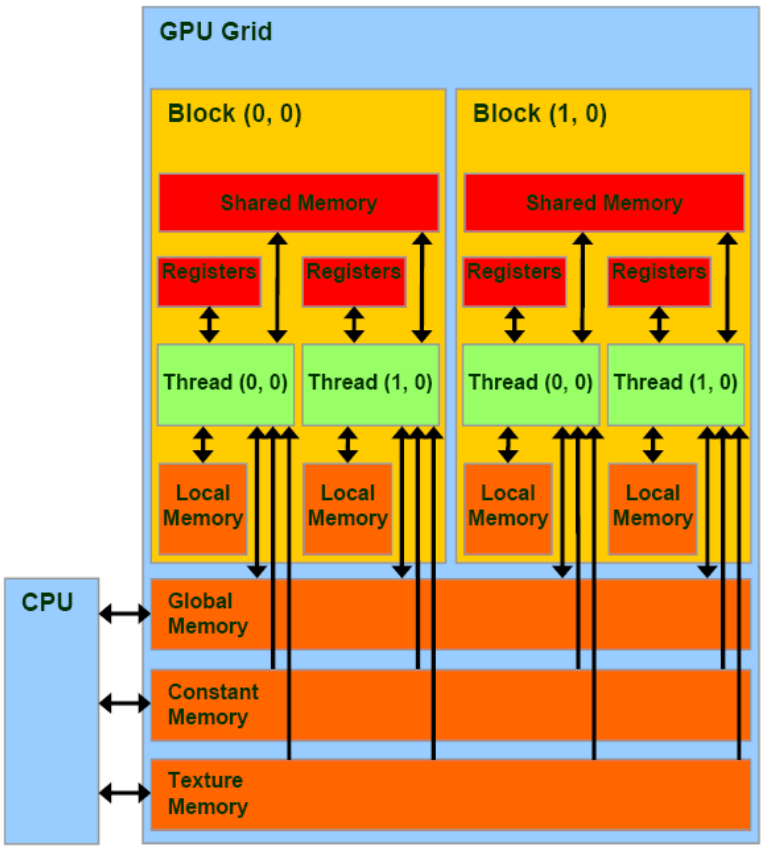

Figura 24: Hierarquia de memória CUDA

A Tabela 3 mostra as características destes tipos de memória:

Tabela 3: Tipos de memória em CUDA

Tipo de Memória	Escopo	Leitura/Escrita	Duração
Global	*device*	Leitura/Escrita	Aplicação
Compartilhada	bloco	Leitura/Escrita	Bloco
Local	*thread*	Leitura/Escrita	*Thread*
Memória de Constantes	*device*	Leitura	Aplicação
Memória de Textura	*device*	Leitura	Aplicação

2.8 CUDA e as aplicações

Hoje em dia pode-se dizer que o modelo de programação para GPUs está na "Era de Ouro" devido a diversos fatores. Entre alguns destes fatores pode-se citar:

- CUDA suporta programação paralela em C diferentemente do modelo anterior de programação que era conhecido como GPGPU, o que facilita o trabalho dos programadores;
- Pesquisadores do mundo todo estão desenvolvendo novos algoritmos e aplicações que se beneficiam da extrema execução de operações de ponto-flutuante das GPUs;
- A comunidade CUDA tem reportado um *speedup* de 10 a 100 vezes de suas aplicações com CUDA. Comparado estes resultados com o avanço da velocidade dos processadores de apenas duas vezes a cada dois anos, verifica-se uma grande diferença.

No campo da Simulação Científica não é diferente, pois muitas pesquisas têm surgido com importantes avanços para a comunidade com o uso desta tecnologia. Alguns motivos relevantes têm levado os pesquisadores a investir nesta tecnologia e entre eles pode-se citar:

- Obtenção de um *speedup* na ordem de uma a duas ordens de magnitude por GPU a um preço baixo até para estudantes;

- Surgimento de um extraordinário número de artigos na literatura desde a introdução de CUDA na comunidade;
- Esta tecnologia é impactante e possui um potencial para afetar fundamentalmente a pesquisa cientifica através da remoção de barreiras de limitação do tempo necessário para se descobrir algo (*time-to-discovery barriers*);
- Saber que as simulações usando esta tecnologia podem ser concluídas em alguns dias e não mais anos como anteriormente;
- Possibilidades de *insights* científicos melhores, pois agora os pesquisadores podem trabalhar com mais dados em paralelo e usar maior precisão;
- O surgimento de uma era onde *clusters* híbridos e supercomputadores contendo um grande número de GPUs estão sendo construídos e utilizados em todo o mundo com custos financeiros cada vez menores;
- Como resultado, muitos pesquisadores (e agências de financiamento) agora tem que repensar seus modelos computacionais e investir em softwares para criar aplicações escaláveis e de alto desempenho baseadas nesta tecnologia.

3 MATERIAIS E MÉTODOS

No decorrer desta seção serão expostos os métodos utilizados para paralelizar as técnicas de escalonamento linear que substituem a etapa de diagonalização e como serão realizados os experimentos.

3.1 Bibliotecas de Matrizes Esparsas

Além de se conhecer os formatos utilizados para o tratamento de matrizes esparsas, também é necessário conhecer como realizar operações sobre estas matrizes. Para facilitar este trabalho, foram buscadas na comunidade científica, bibliotecas para o tratamento de matrizes esparsas. A seleção das bibliotecas, dentre as várias disponíveis, obedeceu alguns critérios como:

- Possuir código aberto;
- Ser liberada para ser implementada integralmente ou parcialmente em outro programa;
- Ter possibilidade de ser modificada para adequação em novos programas;
- Que adote os formatos CSR ou CSC;
- Ser bem documentada;
- Já ter sido utilizada em programas de química quântica.

Dentre todas as bibliotecas encontradas na literatura, as que preencheram a maioria desses requisitos foram as bibliotecas: SPARSKIT2, PSBLAS, Sparse BLASL da MKL e a CUSPARSE. As que serão utilizadas neste trabalho são a SPARSKIT2 e CUSPARSE, em especial a última, pois possui operações que são realizadas nas GPUs.

Depois de escolhidas as bibliotecas, o próximo passo é transformar os dados utilizados no procedimento SCF, de modo que eles estejam num formato apropriado para que se possa trabalhar com eles. O formato escolhido para tratar matrizes esparsas foi o formato CSR.

Sendo assim todas as matrizes usadas nos cálculos no MOPAC foram transformadas para o formato CSR.

3.2 Procedimentos Desenvolvidos

O próximo passo foi programar a multiplicação e o cálculo do traço de matrizes esparsas, que são as principais operações envolvidas nas técnicas de escalonamento linear. Como o objetivo é o ganho de desempenho, estes procedimentos foram desenvolvidos no modelo de programação CUDA para se beneficiar do paralelismo de dados. Para isso foram programados *kernels* para cada procedimento necessário e cada *kernel* desenvolvido em CUDA terá um procedimento respectivo em C responsável por chamá-lo que será chamado de *driver*.

Como o MOPAC está escrito em FORTRAN é necessário fazer um módulo que contenha interfaces que se relacionam com os códigos escritos em CUDA/C. Não há como o código em FORTRAN chamar o *kernel* diretamente, por isso ele invocará o *driver* que será responsável por executar o *kernel*. A interoperabilidade entre CUDA/C e FORTRAN se torna possível através da utilização do módulo ISSO_C_BINDING de FORTRAN. Ou seja, uma interface será responsável por fazer a ligação do código em FORTRAN com o *driver* em C.

3.2.1 Traço da Multiplicação de Matrizes

Um procedimento específico é o cálculo do traço da multiplicação de duas matrizes. O traço de uma matriz é a soma dos elementos da diagonal principal. Um modo de realizar esta operação é calcular a multiplicação e depois calcular o traço. Porém este trabalho pode ser reduzido caso se calcule apenas os elementos da diagonal da matriz resultado da multiplicação. Além disso, esta função tem que suportar o formato CSR e terá que seguir o modelo de programação CUDA, o que modificará um pouco a estrutura normal do algoritmo. O pseudocódigo desta função é mostrado na Figura 25:

```
Traço da Multiplicação de Matrizes(a, ia, ja, b, ib, jb, n)
1.  resultado = 0
2.  do i = 0 to n-1
3.      do j = ia[i] to ia[i+1]-1
4.          col = ja[j]
5.          do k = ib[col] to ib[col+1]-1
6.              col2 = jb[k]
7.              if(col2 == i)
8.                  resultado = resultado + a[j]*b[k]
9.              end if
10.         end do
11.     end do
12. end do
13. return resultado
```

Figura 25: Pseudocódigo do Traço da Multiplicação de Matrizes

Neste pseudocódigo considere que as variáveis *a, ia* e *ja* representam a primeira matriz no formato CSR, as variáveis *b, ib* e *jb* representam a segunda matriz no formato CSR e a variável *n* é a dimensão quadrada das duas matrizes. Como o procedimento em CUDA não pode retornar um valor para o *host*, uma variável que será armazenada na memória global do *device* terá o valor do traço. Uma variável intermediária do tipo __*shared*__ será usada para armazenar os valores parciais do traço. Isso porque o acesso a este tipo de memória é mais rápido do que o acesso a memória global. Porém, como esta variável é compartilhada, é necessário utilizar artifícios, como *mutex*, para a realização de operações atômicas sobre esta variável garantindo que o resultado seja correto. A soma dos resultados parciais é realizada através de um processo chamado *parallel sum reduction tree*. A Figura 26 mostra esta operação:

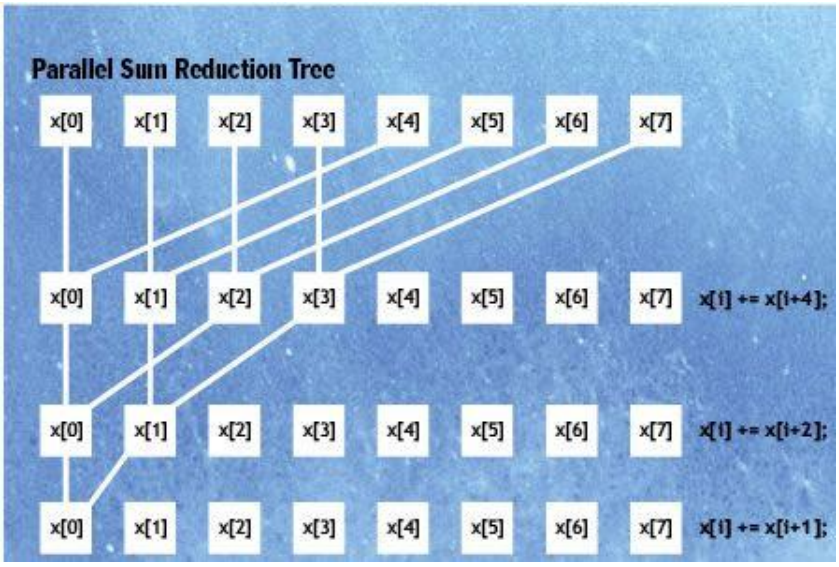

Figura 26 Parallel Sum Reduction Tree

O pseudocódigo desta operação é mostrado na Figura 27.

Sum Reduction
1. *i = blockDim.x/2*
2. *while(i ≠ 0)*
3. *if(id < i)*
4. *output[id] = output[id] + output[id+i]*
5. *end if*
6. i = *i/2*
7. *syncthreads()*
8. *end while*

Figura 27: Exemplo de Redução Paralela de Soma em CUDA

A versão inicial desta operação de cálculo do traço da multiplicação de matrizes utiliza um *mutex* binário para controlar o acesso à variável que conterá o valor do traço. Outra versão utiliza as próprias operações atômicas de CUDA. Estas operações atômicas de soma com números de ponto-flutuante só são possíveis em placas com arquitetura Fermi, e mesmo assim, a dupla precisão ainda não é suportada.

Cada versão utiliza artifícios diferentes para atualizar os valores parciais do traço gerado por cada bloco de *threads.* Na Figura 28 é mostrado como o valor do traço é construído a partir dos valores parciais de cada bloco utilizando um *mutex* e na Figura 29 é

mostrado o pseudocódigo da operação que calcula o traço da multiplicação de matrizes em CUDA:

```
Atualiza Traço()
1.  if(cacheIdx == 0)
2.      mutex.lock()
3.      resultado = resultado + shared_cache[0]
4.      mutex.unlock()
5.  end if
```

Figura 28: Atualiza Traço em CUDA usando *mutex*

```
Traço da Multiplicação de Matrizes em CUDA(a, ia, ja, b, ib, jb, n, resultado)
1.  shared_cache[N_THREADS] = 0
2.  tid = threadIdx.x + (blockIdx.x*blockDim.x)
3.  cacheIdx = threadIx.x
4.  while(tid < n)
5.      do i = ia[tid] to ia[tid+1]-1
6.          col = ja[i]
7.          do j = ib[col] to ib[col+1]-1
8.              col2 = jb[j]
9.              if(col2 == tid)
10.                 shared_cache[cacheIdx] = shared_cache[cacheIdx] + (a[i]*b[j])
11.             end if
12.         end do
13.     end do
14.     tid += gridDim.x*blockDim.x
15. end while
16. synchthreads()
17. Sum-like-tree-reduction(shared_cache, n)
18. Atualiza Traço()
```

Figura 29: Pseudocódigo do Traço de Multiplicação de Matrizes em CUDA

Outro modo de atualizar o valor do traço é utilizando operações atômicas que já vem na API CUDA. Esta operação é mostrada na Figura 30:

```
Atualiza Traço()
1.  if(cacheIdx == 0)
2.      atomicAdd(resultado, shared_cache[0])
3.  end if
```

Figura 30: Atualiza Traço em CUDA usando operações atômicas

O *mutex* binário em CUDA pode ser programado utilizando o pseudocódigo mostrado na Figura 31:

```
Struct Lock
1.  struct Lock {
2.      int *mutex
3.      Construtor() {
4.          AllocEmCUDA(mutex, 1, inteiro)
5.          mutex = 0
6.      }
7.      Destrutor() {
8.          FreeEmCUDA(mutex)
9.      }
10.     __device__  void lock() {
11.         while( atomicCAS(mutex, 0, 1) ≠ 0)
12.     }
13.     __device__ void unlock() {
14.         atomicExch(mutex, 0)
15.     }
16. }
```

Figura 31: Estrutura para simular um *mutex* binário

Como ainda não é suportada a operação atômica para números de ponto flutuante de dupla precisão, esta operação pode ser simulada utilizando o código mostrado na Figura 32:

```
__device__ inline void myAtomicAdd(double *address, double value)
1.  unsigned long long oldval, newval, readback;
```

```
2.  oldval = __double_as_longlong(*address);
3.  newval = __double_as_longlong(__longlong_as_double(oldval) + value);
4.  while((readback=atomicCAS((unsigned long long *)address, oldval, newval)) != oldval)
5.  {
6.      oldval = readback;
7.      newval = __double_as_longlong(__longlong_as_double(oldval) + value);
8.  }
```

Figura 32: Operação atômica de soma para dupla precisão

A operação de atualização do traço, então ficará como é apresentado na Figura 33:

```
Atualiza Traço()
1.  if(cacheIdx == 0)
2.      myAtomicAdd(resultado, shared_cache[0])
3.  end if
```

Figura 33: Atualiza Traço para dupla precisão

3.2.2 Multiplicação de Matrizes

A multiplicação de matrizes esparsas é, sem dúvida, a função mais importante a ser programada, pois nos métodos de escalonamento linear são realizadas várias multiplicações de matrizes que são esparsas. O intuito inicial era programar uma multiplicação que recebia como entrada duas matrizes esparsas no formato CSR e geraria uma matriz esparsa resultante também no formato CSR, tudo isto ocorrendo em um único passo. Porém, na multiplicação de duas matrizes esparsas no formato CSR não se sabe o esqueleto da matriz resultante, ou seja, as dimensões dos *arrays* que compõem o formato CSR. Isto se torna uma limitação para os *kernels* a serem desenvolvidos, pois não há como alocar dinamicamente memória no *device* enquanto se executa o *kernel*. A memória do *device* tem que ser alocada previamente, mas as dimensões dos *arrays* do formato CSR só são definidas durante a execução do *kernel*. Uma alternativa é realizar um procedimento que define as dimensões dos *arrays* previamente sem computar os valores da matriz resultante. O pseudocódigo deste procedimento é mostrado na Figura 34:

Em suma, esta operação de pré-computar as dimensões dos *arrays* da matriz resultante é idêntica à operação de multiplicação de matrizes, se diferenciando apenas porque não realiza multiplicações e adições de elementos.

Esta alternativa, porém, não é a alternativa mais eficiente. Uma segunda alternativa é programar a multiplicação de duas matrizes esparsas no formato CSR para gerar uma matriz no formato denso. Esta alternativa irá consumir mais memória do *device*, mas em compensação vai contornar o problema de alocação dinâmica do *device*. O pseudocódigo desta operação é mostrado na Figura 35:

```
PREAMUB(ia, ja, ib, jb, n, ic)
1.  nnzc = 0
2.  ic[0] = 0
3.  tmp[0...n] = -1
4.  do i = 0 to n-1
5.      do j = ia[i] to ia[i+1]-1
6.          coluna_a = ja[j]
7.          do k = ib[coluna_a] to ib[coluna_a+1]-1
8.              coluna_b = jb[k]
9.              if(tmp[coluna_b] ≠ i)
10.                 nnzc++
11.                 tmp[coluna_b] = i
12.             else
13.                 continue
14.             end if
15.         end do
16.     end do
17.     ic[i+1] = nnzc
18. end do
19. return nnzc
```

Figura 34: Pseudocódigo da PREAMUB de modo serial

Para o código em FORTRAN utilizar a multiplicação de matrizes esparsas da CUSPARSE, foi necessário criar um *driver* que seria responsável por receber duas matrizes esparsas como entrada e gerar outra matriz esparsa como saída. A tarefa de criação do *driver*

envolvia, então, não apenas realizar a multiplicação, mas também envolvia fazer a conversão entre formatos para se adequar a proposta. A CUSPARSE possui procedimentos para a conversão entre formatos e estes procedimentos foram usados na criação do *driver*. O pseudocódigo do *driver* ficou como é apresentado na Figura 36:

```
AMUB(a, ia, ja, b, ib, jb, n, c, mutex)
1.  tid = threadIdx.x
2.  bid = blockIdx.x
3.  i = bidx
4.  while(i < n)
5.      idx = ia[i]
6.      j = tid
7.      while(j+idx < ia[i+1])
8.          coluna_a = ja[j+idx]
9.          do k = ib[coluna_a] to ib[coluna_a+1]-1
10.             coluna_b = jb[k]
11.             produto = a[j+idx]*b[k]
12.             mutex.lock()
13.             shared_cache[coluna_b] += produto
14.             mutex.unlock()
15.         end do
16.         j += blockDim.x
17.     end while
18.     j = tid
19.     syncthreads()
20.     while(j < n)
21.         c[j*n+i] = shared_cache[j]
22.         j += blockDim.x
23.     end while
24.     i += gridDim.x
25. end while
```

Figura 35: Pseudocódigo da AMUB em CUDA

A mult B(a, ia, ja, b, ib, jb, c, ic, jc, n, nnza, nnzb, nnzc)
1. *Inicialização da CUSPARSE*
2. *Configuração da CUSPARSE*
3. *Transformar a matriz B de CSR para o formato DNS com a CUSPARSE*
4. *Multiplicar a matriz A (CSR) pela matriz B(DNS) e gerar a matriz C (DNS) com a CUSPARSE*
5. *Contar o número de elementos não-zeros da matriz C (DNS) com a CUSPARSE*
6. *Transformar a matriz C de DNS para o formato CSR com a CUSPARSE*

Figura 36: Pseudocódigo de A mult B utilizando a CUSPARSE

3.2.3 Métodos de Purificação

Os métodos de Purificação desenvolvidos neste trabalho foram o *Trace Re-setting* e o *Canonical Purification.* Estes métodos foram desenvolvidos tanto para a CPU, como para a GPU, usando tanto o formato denso como o formato esparso (CSR). Para estes métodos serem acrescidos ao MOPAC foi desenvolvido um método SCF simples, ou seja, sem técnicas que aceleram a convergência. Com as operações de cálculo do traço, de adição e de multiplicação de matrizes, o desenvolvimento destes métodos de purificação ficou mais fácil.

3.3 Criação de módulos FORTRAN

Como foi discutido, é necessário criar módulos em FORTRAN que serão responsáveis por chamar os códigos escritos em CUDA/C. Estes módulos devem possuir interfaces que servirão de *link* para os códigos em CUDA/C. Este trabalho pode ser facilitado caso se use o módulo ISSO_C_BINDING de FORTRAN que faz o papel de interoperabilidade entre estas linguagens. Uma descrição mais detalhada deste módulo pode ser encontrada em (Interoperability with C). Aqui só serão apresentados os elementos básicos utilizados para a criação das interfaces propostas. Na Figura 37 será apresentado um modelo base para a criação destas interfaces:

```
module cuda_fortran
1.  interface amub
2.      subroutine amub(a, ia, ja, b, ib, jb, c, ic, jc, n) bind (c, name='amub_c')
3.          use iso_c_binding
4.          implicit none
5.          integer(c_int), value :: n
6.          integer(c_int), dimension(:) :: ia, ja, ib, jb, ic
7.          real(c_double), dimension(:) :: a, b, c
8.      end subroutine
9.  end interface
```

Figura 37: Exemplo de interface em FORTRAN para um *driver* em C

O protótipo em C deste procedimento é mostrado na Figura 38:

```
extern "C" void amub_c(double *a, int *ia, int *ja, double *b, int *ib, int *jb, double *c,
                       int *ic, int *jc, int n);
```

Figura 38: Exemplo do protótipo em C do procedimento A mult B

3.4 Modificação do MOPAC

O MOPAC utiliza o conceito de *keywords* para indicar o que será calculado e quais métodos serão usados. Isto facilitou a incorporação dos novos métodos produzidos, pois foram acrescentadas novas *keywords* para que estes novos métodos desenvolvidos fossem utilizados. Assim a estrutura funcional do MOPAC não foi alterada e as *keywords* possibilitaram esta flexibilidade. As seguintes *keywords* mostradas na Tabela 4 foram criadas:

Tabela 4: Keywords acrescentadas ao MOPAC

Mopac Keywords	**Description**
RUNGPU	Cálculo do MOPAC usando GPU
DIAGLAPACK	Uso da LAPACK para a diagonalização
DBGGPU	Opção de Debug para RUNGPU
CALCPMETH=	• Se 1, calcula a matriz de densidade na GPU sem cublas; • Se 2, calcula a matriz de densidade na GPU com cublas; • Se 3, calcula a matriz de densidade na CPU com dgemm.
GPUBLOCKS	Número de blocos de *threads* para o *grid* GPU
GPUTHREADS	Número de *threads* por bloco para o *grid* GPU

3.5 Metodologia

O experimento a ser realizado consiste na avaliação das operações paralelas criadas usando CUDA e sua utilização nos métodos de escalonamento linear. Os métodos de escalonamento linear desenvolvidos foram o CG-DMS e algumas versões da Purificação de matrizes de densidade.

A escolha das moléculas foi limitada pela quantidade de recursos e memória a serem alocadas na GPU. O número mínimo de *bytes* a serem armazenados foi determinado pela operação de multiplicação de matrizes utilizando a CUSPARSE. Nesta operação são utilizadas duas matrizes no formato denso e uma matriz no formato esparso. Como o formato denso precisa armazenar n^2 elementos de precisão dupla (8 bytes) e são necessários espaço para armazenar duas matrizes densas, então a memória global mínima a ser utilizada pela GPU é calculada da seguinte forma: $(n)^2 * 16$, onde n é a dimensão da matriz.

Na Tabela 5 são mostradas as moléculas utilizadas para análise de desempenho das operações citadas com suas respectivas dimensões, a quantidade mínima de elementos das matrizes e a quantidade de memória mínima a ser utilizada pela GPU:

Tabela 5: Memória a ser utilizada

Molécula	Dimensão das matrizes	Quantidade mínima de Elementos	Memória a ser utilizada (Gb)
Water Box 80	480	230400	0.00343
Water Box 120	720	518400	0.00772
Water Box 160	960	921600	0.01373
Water Box 300	1800	3240000	0.04828
Water Box 1000	6000	36000000	0.53644

Para os métodos de Purificação, foram utilizados sistemas moleculares contendo moléculas de água organizadas em formato esférico, com raios iguais a 6, 8, 10 e 12 angström, respectivamente. A quantidade de átomos, de orbitais e de moléculas de água para cada modelo é mostrada na Tabela 6:

Tabela 6: Moléculas utilizadas nos métodos de Purificação

Modelo	Átomos	Orbitais	Moléculas de Água
W_6	96	192	32
W_8	228	456	76
W_{10}	417	834	139
W_{12}	711	1422	237

Os testes foram realizados no seguinte ambiente computacional:

- Processador: Intel® Core(TM) i7 920 2.67 GHz;
- Memória: 4 memórias DDR3 com capacidade de 2Gb;
- GPU: GeForce GTX 480 com 1.5Gb de memória.
- Sistema Operacional: Ubuntu 10.04

3.6 Análise de Desempenho

Para que se pudesse analisar o desempenho dos códigos desenvolvidos, foi utilizado um temporizador em CUDA para medir o tempo gasto pelo *kernel* em CUDA. Estes valores foram comparados com os tempos gastos pela realização das mesmas operações de modo serial na CPU e colocados numa tabela.

Como cada chamada do *kernel* em CUDA se diferencia pelos valores que especificam o *grid* em CUDA, foram realizadas várias chamadas a estes *kernels* modificando-se os valores dos parâmetros. Os valores obtidos foram colocados num gráfico para mostrar a diferença das chamadas. O número de *threads* por bloco assumiu valores múltiplos de 32 e potências de 2, respectivamente, pois um *warp* é composto por 32 *threads* e a *parallel sum reduction tree* necessita que a quantidade de *threads* seja potência de 2.

Na realização da multiplicação de matrizes esparsas, porém, não precisou se especificar várias chamadas, pois foi utilizada a CUSPARSE e os parâmetros de *kernel* de suas operações não podem ser especificados. Então foi utilizado apenas o temporizador para marcar o tempo de execução.

As operações analisadas foram: o cálculo do traço da multiplicação de duas matrizes em CUDA e de modo serial na CPU, e a multiplicação de matrizes usando a CUSPARSE e de modo serial na CPU, utilizando a PREAMUB e AMUB. Estas operações utilizaram tanto dados de ponto flutuante de precisão simples como de dupla precisão.

Para os métodos de escalonamento linear foram realizadas iterações SCF usando o método tradicional do MOPAC, a diagonalização, e usando os diferentes métodos desenvolvidos para substituir a diagonalização.

4 RESULTADOS E DISCUSSÕES

Os resultados dos experimentos foram divididos em quatro etapas: a comparação dos *kernels* que calculam o traço da multiplicação de matrizes, a comparação dos tempos obtidos nas operações paralelas na GPU e seriais na CPU, a análise da esparsidade das matrizes utilizadas no cálculo e análise dos tempos obtidos nas iterações SCF.

Os resultados obtidos com as quatro versões da operação de cálculo do traço da multiplicação de matrizes para cada molécula serão apresentados nos gráficos a seguir. Para estes gráficos considere que:

- O *myAtomicAdd* é a operação atômica de soma para números de ponto flutuante de dupla precisão que não é padrão da NVIDIA;
- A operação *atomicAdd* é a operação atômica de soma padrão da API CUDA que só pode ser usada para números de ponto flutuante de precisão simples;
- A operação com *Lock* utiliza a estrutura citada no método proposto para simular o *mutex* e assim garantir acesso à região crítica.

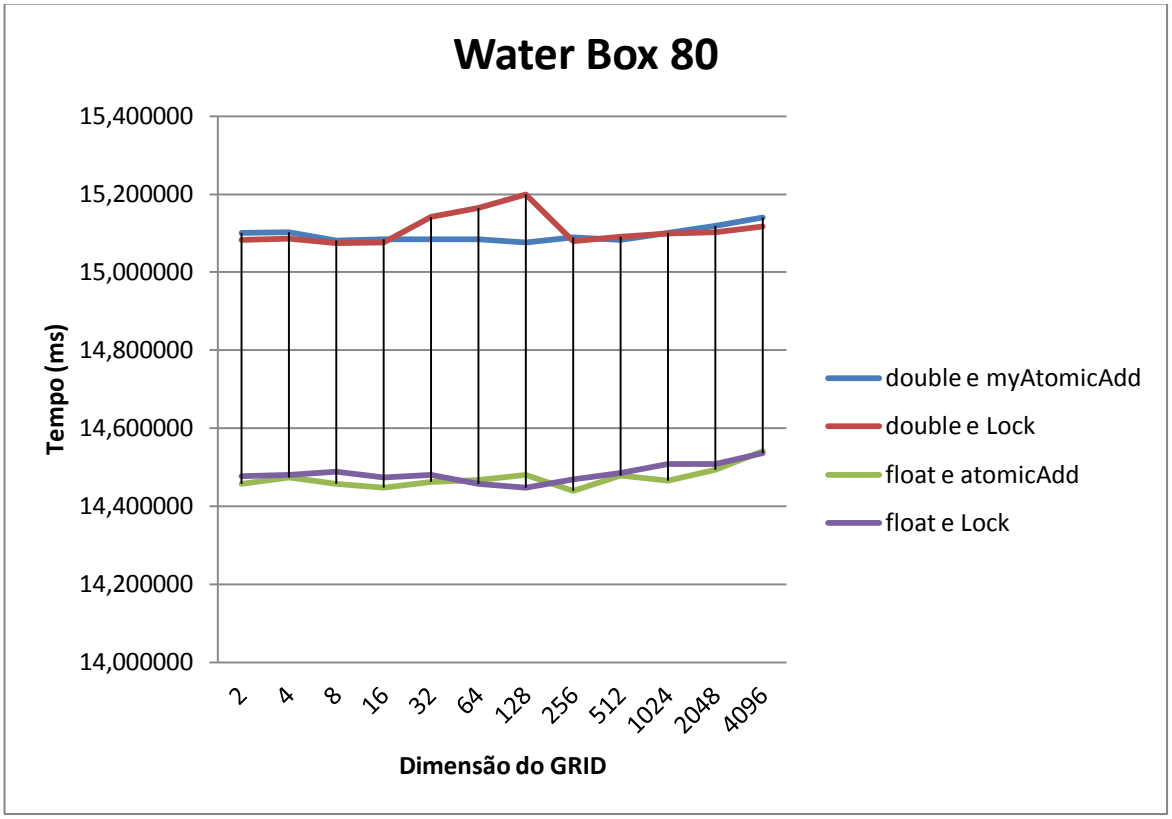

Figura 39: Tempos para a molécula Water Box 80

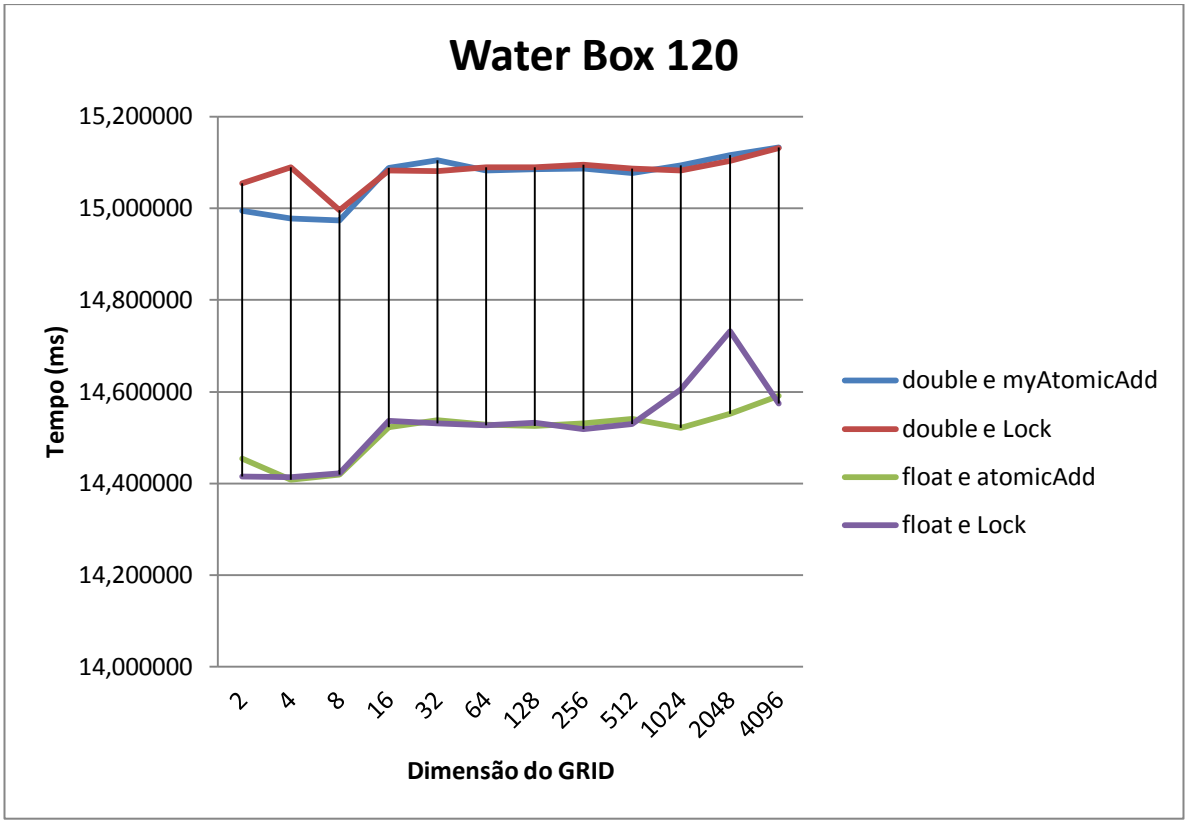

Figura 40: Tempos para a molécula Water Box 120

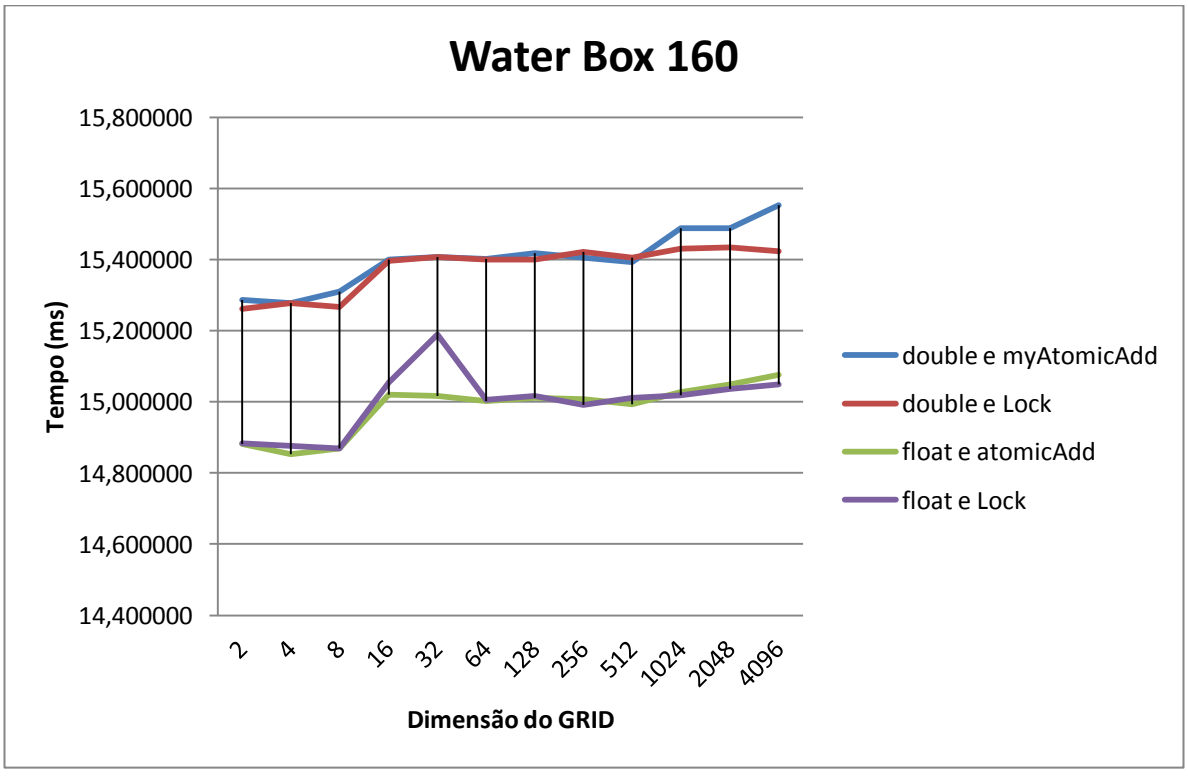

Figura 41: Tempos para a molécula Water Box 160

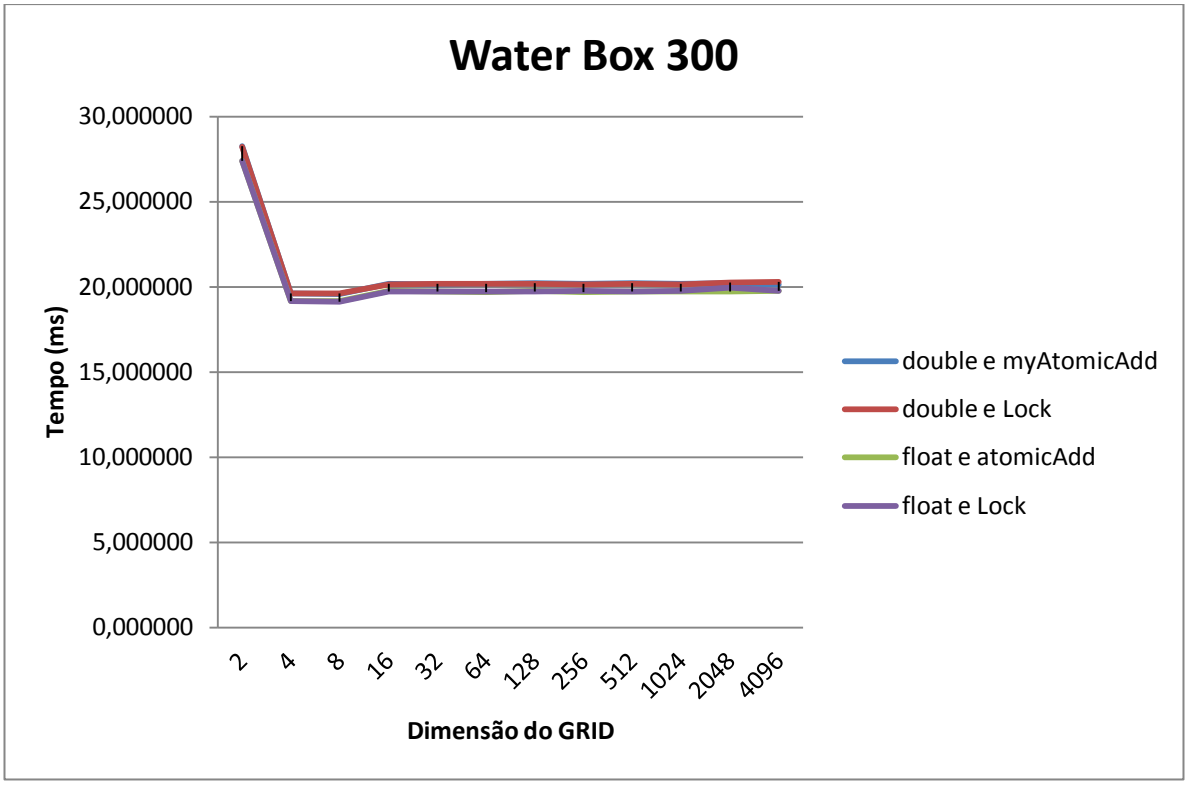

Figura 42: Tempos para a molécula Water Box 300

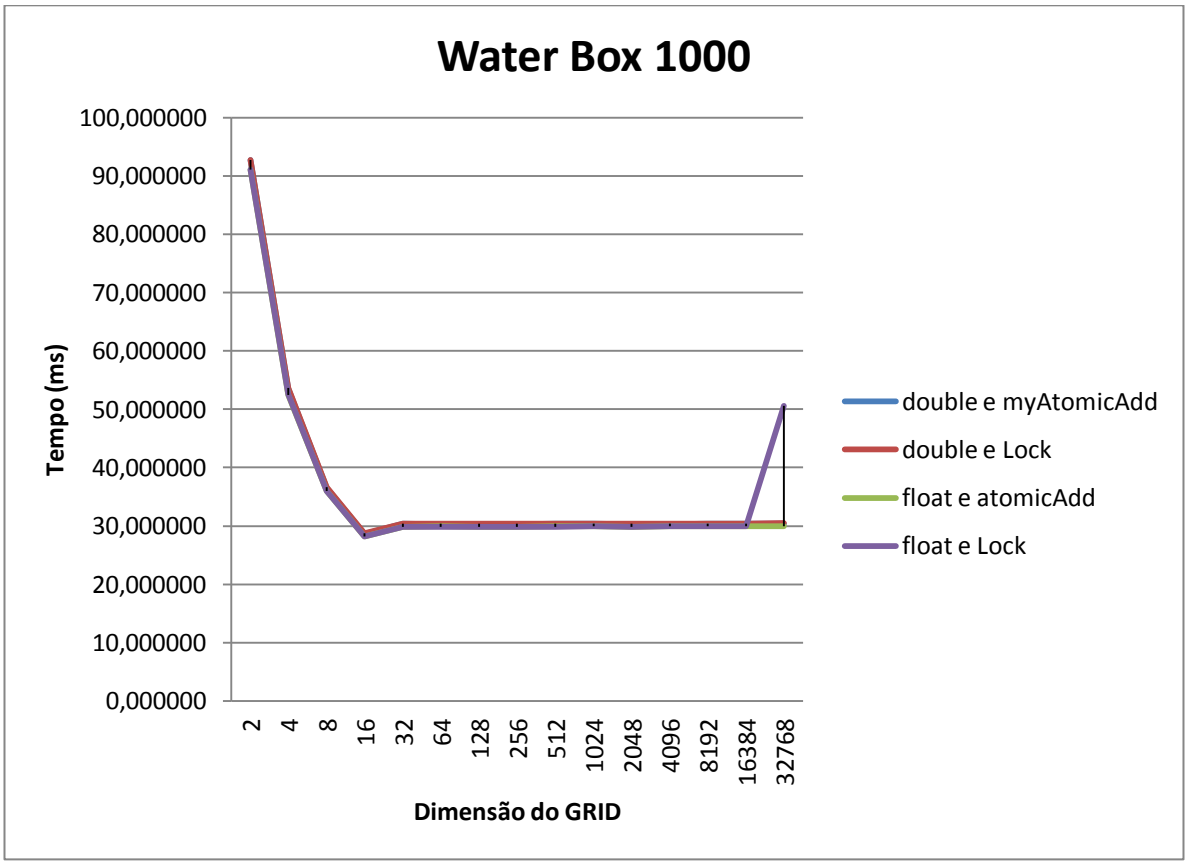

Figura 43: Tempos para a molécula Water Box 1000

A partir da análise destes gráficos, percebe-se que as versões para o cálculo do traço em CUDA apresentaram resultados bem próximos uns dos outros variando apenas quando a dimensão do *grid* é grande. Como era de se esperar os tempos do cálculo com dupla precisão foram maiores, mas a diferença não chega a ser tão impactante quando comparado ao tempo gasto com precisão simples. Outra implicação da análise dos gráficos é que a dimensão do *grid* e do número de *threads* são parâmetros que devem ser levado em conta ao se programar *kernels* em CUDA, pois estes valores influenciam diretamente no desempenho da aplicação.

Vale também informar que o número máximo de *threads* que puderam ser alocados com o código para o cálculo do traço foi 512, sendo que o número máximo é 1024 para esta GPU, em questão. Isto se deve ao fato de que a GPU tem um conjunto limitado de recursos como registradores, memória de constantes e memória compartilhada. Quando os recursos previstos no código não podem ser alocados, então o *kernel* não é lançado e gera um erro conhecido como "*unspecified launch failure*".

Na segunda etapa de resultados é mostrada a comparação entre os métodos na GPU e na CPU. Na Tabela 7 são mostrados os tempos obtidos com a realização da operação de cálculo do traço da multiplicação de matrizes:

Tabela 7: Tempos obtidos com o traço da multiplicação de matrizes

Molécula	GPU (ms)	CPU (ms)
Water Box 80	15,075328	22,739887
Water Box 120	14,973824	32,042027
Water Box 160	15,265952	36,793947
Water Box 300	19,590464	83,682060
Water Box 1000	28,676704	312,453985

Analisando esta tabela se vê que na última instância, que contém um número maior de átomos, obteve-se um *speedup* de aproximadamente 10,9 vezes, enquanto que nas outras instâncias o *speedup* obtido foi inferior a cinco vezes, mas mesmo assim a operação realizada na GPU foi melhor do que a mesma operação realizada na CPU.

A outra operação analisada foi a multiplicação de matrizes. A operação de multiplicação de matrizes na GPU utiliza a CUSPARSE, enquanto que, na CPU utiliza a PREAMUB e AMUB. Os valores obtidos se encontram na Tabela 8:

Tabela 8: Tempos obtidos (em ms) com a multiplicação de matrizes

Molécula	PREAMUB-CPU	AMUB-CPU	AMUB-CUSPARSE
Water Box 80	27,639866	603,318214	5,300512
Water Box 120	39,359808	858,039856	8,298432
Water Box 160	45,272112	988,450050	11,915296
Water Box 300	104,067087	2303,211689	33,165279
Water Box 1000	390,066862	9216,458797	307,376923

Os resultados mostrados aqui são mais impressionantes, pois o *speedup* mínimo alcançado é de 31,25 vezes. Em todas as instâncias a operação de multiplicação de matrizes realizada na GPU é mais rápida do que na CPU, inclusive é mais rápida do que a operação de pré-computar (PREAMUB) as dimensões dos vetores da matriz resultante. A instância que alcançou o maior *speedup* foi a primeira instância cujo *speedup* foi de aproximadamente 119 vezes.

É importante ver também que o tempo gasto na multiplicação de matrizes na CPU é resultado da soma do tempo da operação PREAMUB com a própria AMUB. A vantagem desta abordagem é definir logo quais as dimensões dos *arrays* que compõem o formato CSR da matriz resultante evitando consumo excessivo de memória.

A desvantagem da multiplicação de matrizes realizada com a CUSPARSE é que esta operação utiliza duas matrizes no formato denso consumindo bastante memória da GPU. Uma alternativa é construir uma operação que utilize apenas uma ou nenhuma matriz no formato denso, pois a GPU tem recursos limitados.

As porcentagens de esparsidade das matrizes de Fock geradas nos cálculos SCF e utilizadas nas operações descritas acima se encontram na Tabela 9

Tabela 9: Porcentagens de esparsidade das matrizes de Fock

Molécula	F	F*F
Water Box 80	79,22482639	28,17881944
Water Box 120	86,73070988	43,92052469
Water Box 160	90,69574653	56,14963108
Water Box 300	94,55617284	71,32126543
Water Box 1000	98,25688333	89,57638611

Como se observa, a esparsidade das matrizes cresce à medida que se aumenta o número de átomos da molécula nestes exemplos e isto é uma vantagem quando são utilizadas operações de álgebra linear para matrizes esparsas.

A multiplicação de duas matrizes esparsas, em geral, produz uma matriz menos esparsa e isto tem que ser levado em consideração, pois este crescimento da matriz vai implicar em um consumo maior de memória necessária para armazenar os elementos diferentes de zero da matriz. Duas alternativas podem ser tomadas para diminuir este crescimento: utilização de um filtro após cada operação ou fixação de um esqueleto para armazenar a matriz.

A aplicação do filtro pode eliminar elementos desprezíveis que venham surgir durante as operações aumentando assim a esparsidade da matriz. Pode ser especificado um limiar e todo elemento da matriz que estiver abaixo deste limiar será desprezado.

A fixação de um esqueleto consiste em estabelecer uma forma generalizada para todas as matrizes esparsas dos cálculos. Ou seja, serão armazenados apenas os valores de cada matriz, mesmo sendo estes valores iguais a zero (o *array A* do formato CSR), pois os outros *arrays* (*IA* e *JA* do formato CSR) serão idênticos e únicos para todas as matrizes.

Mesmo que o *speedup* alcançado nestas operações seja pequeno, isto poderá implicar em um grande avanço, pois em cálculos que envolvem dinâmica molecular, por exemplo, tais operações serão realizadas milhões de vezes. Ou seja, qualquer pequeno avanço representará no cálculo todo, uma enorme diferença.

Para os métodos de Purificação, os resultados obtidos para um cálculo completo SCF serão mostrados nas tabelas posteriores. As moléculas utilizadas foram caixas de água de simulação de formato esférico com o valor do raio variando entre 6 e 12 angström. As tabelas apresentam o formato de matriz utilizado (DNS – denso ou CSR – esparso), o método de purificação utilizado (1 – Canonical Purification ou 2 – Trace Re-setting), os tempos obtidos na CPU, os tempos obtidos na GPU e o tempo de referência, que consiste na utilização do MOPAC sem nenhuma alteração.

Tabela 10: Resultados obtidos com a caixa de simulação de raio 6Å

Formato da Matriz	Método de Purificação	Tempo na CPU(s)	Tempo na GPU(s)	Tempo de Referência(s)
DNS	1	2.80	5.23	0.30
	2	2.85	5.51	
CSR	1	10.20	5.16	
	2	13.49	8.61	

Tabela 11: Resultados obtidos com a caixa de simulação de raio 8Å

Formato da Matriz	Método de Purificação	Tempo na CPU(s)	Tempo na GPU(s)	Tempo de Referência(s)
DNS	1	17.98	15.94	2.30
	2	18.50	16.59	
CSR	1	112.71	12.11	
	2	163.21	30.34	

Tabela 12: Resultados obtidos com a caixa de simulação de raio 10Å

Formato da Matriz	Método de Purificação	Tempo na CPU(s)	Tempo na GPU(s)	Tempo de Referência(s)
DNS	1	78.88	49.92	13.32
	2	83.42	49.10	
CSR	1	663.19	30.71	
	2	962.95	84.84	

Tabela 13: Resultados obtidos com a caixa de simulação de raio 12Å

Formato da Matriz	Método de Purificação	Tempo na CPU(s)	Tempo na GPU(s)	Tempo de Referência(s)
DNS	1	1744.99	126.63	153.02
	2	2990.12	131.00	
CSR	1	3137.65	79.80	
	2	4380.00	223.40	

Analisando os resultados das tabelas, percebe-se que entre os métodos de purificação desenvolvidos, o que obteve os melhores resultados para estas moléculas foi o método *Canonical Purification*. Outro detalhe a ser analisado, é que na implementação dos métodos

para a CPU, aqueles que utilizaram o formato denso foram melhores do que aqueles que utilizaram o formato esparso. Isto se deve ao fato de que as operações esparsas na CPU utilizaram a SPARSEKIT 2, cujo desempenho ainda não é tão satisfatório para matrizes esparsas. As operações no formato denso utilizaram as sub-rotinas da BLAS.

Os tempos obtidos pela CPU para os métodos de purificação foram inferiores ao tempo de referência, pois o método SCF utilizado pelo MOPAC é mais robusto, possuindo aceleradores de convergência que não puderam ser utilizados nos métodos de purificação. Sendo assim o SCF para os métodos de purificação foi realizado de forma simplificada, sem aceleradores afetando o desempenho.

Com relação aos tempo obtidos na GPU, observa-se que foram melhores em todas as instâncias do que os tempos obtidos na CPU, exceto para a molécula de raio 6, cujos tempos utilizando o formato denso foram piores. Os tempos obtidos utilizando o formato esparso foram melhores do que os tempos obtidos utilizando o formato denso, apenas para o método *Canonical Purification.* Nos outros casos, estes tempos foram inferiores. Outro fato a ser observado foi que na molécula de raio 12, o tempo obtido no formato esparso e utilizando o método *Canonical Purification* foi melhor do que o tempo de referência, mesmo utilizando um SCF menos robusto.

Se for calculado o *speedup* da aplicação para o método *Canonical Purification* utilizando o formato esparso, percebe-se que ele aumenta à medida que o tamanho da molécula aumenta. O menor *speedup* calculado foi aproximadamente 2 vezes e o maior foi aproximadamente 39 vezes.

Entre as principais dificuldades encontradas destacam-se as limitações do *hardware* da GPU e a dificuldade de portar todo o código desenvolvido para o MOPAC. Como o *hardware* das GPUs ainda está em constante crescimento, ele possui algumas limitações que impedem o desenvolvimento da aplicação. A principal limitação é a memória, pois nos casos em que se utilizam matrizes no formato denso, muitas vezes não há memória suficiente para realizar cálculos sobre moléculas com milhares de átomos. A operação que mais é influenciada por esta limitação é a multiplicação de matrizes esparsas, pois como já foi discutido na seção que descreve o método proposto, as dimensões dos *arrays* utilizados nestas operações só são determinadas em tempo de execução e a alocação de memória no *device* tem que ser realizada previamente. Alocar a matriz no formato denso na memória do *device* é um desperdício de memória.

Em relação ao desempenho dos algoritmos desenvolvidos, algumas questões como utilização dos diferentes tipos de memória do *device* influenciaram drasticamente o

desempenho da aplicação. Por exemplo, utilizar memória compartilhada (*shared*) no lugar da memória global influencia no desempenho da aplicação, sabendo-se que a memória compartilhada é mais escassa e nem sempre pode ser utilizada para todo propósito. É necessário conhecer bem a arquitetura CUDA para tirar o melhor proveito e desempenho.

5 CONCLUSÃO

Ao fim deste trabalho, conclui-se que a Computação de Alto Desempenho aplicada à Química Quântica possibilita novos avanços na indústria farmacêutica. Isto porque cada vez mais novos compostos químicos complexos, como proteínas e enzimas, podem ser estudados e sintetizados com maior facilidade e mais rapidez.

A Computação de Alto Desempenho utilizando o modelo CUDA de programação foi responsável por este grande avanço, pois permitiu que os métodos utilizados na Química Quântica pudessem ser paralelizados, dividindo o esforço necessário para a realização dos cálculos destes métodos entre os vários núcleos de processamento das placas gráficas.

O modelo de programação CUDA é, sem dúvida, uma alternativa para programação massiva paralela de dados, pois necessita utilizar poucos recursos se comparado aos recursos gastos em *clusters.* Isto se torna mais um incentivo para que programadores e pesquisadores venham investir e adaptar suas aplicações a este novo modelo de programação.

A utilização de Bibliotecas de Matrizes Esparsas também desempenhou papel fundamental no desenvolvimento de métodos paralelos de alto desempenho, pois eliminou cálculos que não eram necessários. Os formatos utilizados para representar as matrizes esparsas neste trabalho eliminaram a necessidade de armazenar elementos com valores próximos de zero, aumentando o desempenho dos cálculos.

Como foram descritos na seção de Resultados e Discussões, os tempos obtidos nas operações desenvolvidas estão na escala de milissegundos, o que pode implicar um pequeno avanço. Porém vale ressaltar que estas operações são realizadas milhões de vezes em alguns cálculos como dinâmica molecular, transformando estes pequenos avanços em grandes avanços.

A tarefa de paralelizar o código do MOPAC foi crucial para a efetivação deste trabalho. O código do MOPAC foi desenvolvido antes da “Era da Computação Paralela”, por isso é todo seqüencial e a sua reescrita de modo paralelo é uma tarefa que exige bastante dedicação considerando que ele tem mais de 30.000 linhas de código. Nestas linhas de código encontram-se códigos obsoletos e códigos escritos por vários programadores, cada um com seu estilo de codificação, tornando muito difícil seu completo entendimento. Todos estes fatores devem ser levados em conta quando se quer paralelizar códigos legado como o MOPAC.

Estes avanços mostrados neste trabalho só foram possíveis depois de um estudo aprofundado da teoria que envolve estes métodos para identificação de necessidades e possíveis melhorias. Isto se torna um desafio, principalmente porque envolve o estudo de uma área que não é comum para os cientistas da computação.

Este trabalho também foi importante para mostrar que a ciência da computação é uma ciência que pode ajudar as mais diversas ciências, auxiliando na computação de dados de forma mais otimizada e sendo responsável por acelerar os métodos existentes nas ciências.

A continuação destes estudos poderá proporcionar o melhor desempenho de novos métodos e dos métodos já existentes que não foram cobertos por este trabalho, pois o desenvolvimento de métodos de escalonamento linear está em constante renovação e aperfeiçoamento e o paralelismo ainda não está presente na maioria destes métodos. É preciso aplicar a metodologia da Computação de Alto Desempenho para que as pesquisas desta área venham alcançar novos horizontes e avanços.

Como trabalhos futuros que surgiram como frutos deste trabalho pode se destacar o desenvolvimento de um *software* específico voltado para ser utilizado em computadores paralelos e com GPUs com capacidade CUDA que irá conter novas técnicas de escalonamento linear que serão desenvolvidas a fim de que se consiga obter um melhor desempenho em diversos cálculos realizados no campo da Química Quântica.

REFERÊNCIAS

ANIKIN, N. A. et al. LocalSCF method for semiempirical quantum-chemical calculation of ultralarge biomolecules. **Journal of Chemical Physics**, v. 121, n. 3, p. 1266-1270, Julho 2004.

BROYDEN, C. G. The Convergence of a Class of Double-rank Minimization Algorithms 1. General Considerations. **IMA Journal of Applied Mathematics**, v. 6, n. 1, p. 76-90, 1970.

COMPUTATIONAL Chemistry. **NVIDIA**. Disponivel em: <http://www.nvidia.com/object/computational_chemistry.html>. Acesso em: 11 fevereiro 2012.

DANIELS, A. D.; MILLAM, J. M.; SCUSERIA, G. E. Semiempirical methods with conjugate gradient density matrix search to replace diagonalization for molecular systems containing thousands of atoms. **Journal of Chemical Physics**, v. 107, p. 425-432, Julho 1997.

DANIELS, A. D.; SCUSERIA, G. E. What is the best alternative to diagonalization of the Hamiltonian in large scale semiempirical calculations? **Journal of Chemical Physics**, v. 110, n. 3, p. 1321-1328, Janeiro 1999.

DE JONG, W. A. et al. Utilizing high performance computing for chemistry: parallel computational chemistry. **Physical Chemistry Chemical Physics**, v. 12, p. 6896-6920, 2010.

DEWAR, M. J. S.; HEALY, E. F. AM1: A New Genearl Purpose Quantum Mechanical Molecular Model. **Journal of the American Society**, v. 107, n. 13, p. 3902-3909, 1985.

DEWAR, M. J. S.; THIEL, W. Ground state of molecules. 38. The MNDO method. Approximations and parameters. **Journal of the American Chemical Society**, v. 99, n. 15, p. 4899-4907, 1977.

FILIPPONE, S.; COLAJANNI, M. PSBLAS: a library for parallel linear algebra linear algebra computation on sparse matrices. **Journal ACM Transactions on Mathematical Software**, v. 26, n. 4, p. 527-550, 2000.

GOEDECKER, S.; SCUSERIA, G. E. Linear Scaling Eletronic Structure Methods in Chemistry and Physics. **Computational Chemistry**, p. 14-21, 2003.

HAMILTON, T. P.; PULAY, P. Direct inversion in the iterative subespace (DIIS) optimization of open-shell, excited-state, and small multiconfigurational SCF wave function. **Journal of Chemical Physics**, v. 84, p. 5728-5734, 1986.

INTEROPERABILITY with C. **BCS Fortran Specialist Group**. Disponivel em: <http://www.fortran.bcs.org/2002/interop.htm>. Acesso em: Julho 2011.

JANSSEN, C. L.; NIELSEN, I. M. B. **Parallel Computing in Quantum Chemistry**. [S.l.]: CRC Press, 2008.

KOHN, W.; SHAM, L. J. Self-Consistent Equations Including Exchange and Correlation Effects. **Physical Review**, v. 140, p. 1133-1138, 1965.

LEE, T.-S.; YORK, D. M.; YANG, W. Linear-scaling semiempirical quantum calculations for macromolecules. **Journal of Chemical Physics**, v. 105, n. 7, p. 2744-2750, Agosto 1996.

LIN, T.-H. **Parallelizing Legacy Applications using Message Passing Programming Model and the Example of MOPAC**. Graduate School of Syracuse University. [S.l.]. 2000.

MASLEN, P. E. et al. Locality and Sparsity of Ab Initio One-Particle Density Matrices and Localized Orbitals. **The Journal of Physical Chemistry**, v. 102, n. 12, p. 2215-2222, 1998.

MATH Kernel Library from Intel - Intel Software Network. Disponivel em: <http://software.intel.com/en-us/articles/intel-mkl/>. Acesso em: agosto 2011.

MILLAM, J. M.; SCUSERIA, G. E. Linear scaling conjugate gradient density matrix search as an alternative to diagonalization for first principles eletronic structure calculations. **Journal of Chemical Physics**, v. 106, n. 5569, 1997.

NAUMOV, M. et al. **CUSPARSE LIBRARY**. GPU Technology Conference. San Jose: NVIDIA. 2010.

NEEDHAM, P. **An Acceleration of MUSE using GPU**. The University of Endiburgh. [S.l.]. 2010.

NIKLASSON, A. M. N. Expansion algorithm for the density matrix. **Physical Review B**, v. 66, n. 15, p. 155115, 2002.

NIKLASSON, A. M. N.; TYMCZAK, C. J.; CHALLACOMBE, M. Trace Re-Setting Density Matrix Purification in O(N) Sel-Consistent-Field Theory. **The Journal of Chemical Physics**, v. 118, n. 19, p. 8611-8621, 2003.

NVIDIA. **CUDA C BEST PRACTICES GUIDE**. [S.l.]: [s.n.], v. 4, 2011.

NVIDIA Home. Disponivel em: <http://www.nvidia.com.br/page/home.html>. Acesso em: Julho 2011.

OFFICIAL Gaussian Website. Disponivel em: <http://www.gaussian.com/>. Acesso em: Julho 2011.

PALSER, A. H. R.; MANOLOPOULOS, D. E. Canonical purification of the density matrix in eletronic-structure theory. **Physical Review B**, v. 58, n. 19, p. 12704-12711, November 1998.

PISSANETSKY, S. **Sparse Matrix Technology**. Bariloche: Academic Press, 1984.

ROCHA, G. B. et al. RM1: A Reparameterization of AM1 for H, C, N, O, P, S, F, Cl, Br, and I. **Journal of Computational Chemistry**, v. 27, n. 10, p. 1101-1111, 2006.

ROOTHAAN, C. C. J. New Developments in Molecular Orbital Theory. **Reviews of Modern Physics**, v. 23, n. 2, p. 69-89, Abril 1951.

RUBENSSON, E. **Sparse Matrices for Quantum Chemistry Applications**. Royal Institute of Techonology. [S.l.]. 2005.

RUBENSSON, E. H.; RUDBERG, E.; SALEK, P. A Hierarchic Sparse Matrix Data Structure for Large-Scale Hartre-Fock/Kohn-Sahm Calculations. **Journal of Computational Chemistry**, v. 28, n. 16, p. 2531-2537, 2007.

RUDBERG, E.; RUBENSSON, E. H. Assessment of density matrix methods for linear scaling eletronic structure calculations. **Journal of Physics: Condensed Matter**, v. 23, 2011.

RUDBERG, E.; RUBENSSON, E. H.; SALEK, P. Hartree-Fock calculations with linearly scaling memory usage. **The Journal of Chemical Physics**, v. 128, 2008. ISSN DOI: 10.1063/1.2918357.

SAAD, Y. **SPARSKIT: A basic tool kit for sparse matrix computations**. Research Institute for Advanced Computer Science. [S.l.]. 1990.

SANDERS, J.; KANDROT, E. **CUDA by Example**. [S.l.]: [s.n.], 2010.

STEWART Computational Chemistry - MOPAC - Home Page. Disponivel em: <http://openmopac.net/>. Acesso em: Julho 2011.

STEWART, J. J. P. Optimization of parameters for semiempirical methods V: Modification of NDDO approxiamtions and application to 70 elements. **J. Mol. Modeling**, v. 13, p. 1173-1213, 2007.

STEWART, J. J. P.; CSÁSZÁR, P.; PULAY, P. Fast Semiempirical Calculations. **Journal of Computational Chemistry**, v. 3, n. 2, p. 227-228, 1982.

THIEL, W. **Linear scaling conjugate gradient density matris search: Implementation, validation, and application with semiempirical molecular orbital methods**. [S.l.]. 2003.

VAN DER VAART, A. et al. Linear Scaling Molecular Orbital Calculations of Biological Systems Using the Semiempirical Divide and Conquer Method. **Journal of Computational Chemistry**, v. 21, n. 16, p. 1494-1504, 2000.

WHAT is CUDA | NVIDIA Developer Zone. Disponivel em: <http://developer.nvidia.com/what-cuda>. Acesso em: Julho 2011.