Universidade Federal de Minas Gerais
Escola de Engenharia
Programa de Pós-Graduação em Engenharia Elétrica

Paralelização do método Meshless Local Petrov-Galerkin (MLPG) utilizando processadores gráficos (GPU) e CUDA

Bruno Carvalho Correa

Orientador: Prof. Renato Cardoso Mesquita

Belo Horizonte, novembro de 2013

Universidade Federal de Minas Gerais

Escola de Engenharia

Programa de Pós-Graduação em Engenharia Elétrica

PARALELIZAÇÃO DO MÉTODO MESHLESS LOCAL PETROV-GALERKIN (MLPG) UTILIZANDO PROCESSADORES GRÁFICOS (GPU) E CUDA

Bruno Carvalho Correa

Dissertação de Mestrado submetida à Banca Examinadora designada pelo Colegiado do Programa de Pós-Graduação em Engenharia Elétrica da Escola de Engenharia da Universidade Federal de Minas Gerais, como requisito para obtenção do Título de Mestre em Engenharia Elétrica.

Orientador: Prof. Renato Cardoso Mesquita

Belo Horizonte - MG

Fevereiro de 2014

"Paralelização do Método Meshless Local Petrov-Galerkin (MLPG) Utilizando Processadores Gráficos (GPU) e Cuda"

Bruno Carvalho Correa

Dissertação de Mestrado submetida à Banca Examinadora designada pelo Colegiado do Programa de Pós-Graduação em Engenharia Elétrica da Escola de Engenharia da Universidade Federal de Minas Gerais, como requisito para obtenção do grau de Mestre em Engenharia Elétrica.

Aprovada em 24 de fevereiro de 2014.

Por:

Prof. Dr. Renato Cardoso Mesquita
DEE (UFMG) - Orientador

Prof. Dr. Elson José da Silva
DEE (UFMG)

Prof. Dr. Ricardo Luiz da Silva Adriano
DEE (UFMG)

Prof. Dr. Rodney Rezende Saldanha
DEE (UFMG)

DISSERTAÇÃO DE MESTRADO Nº 814

PARALELIZAÇÃO DO MÉTODO MESHLESS LOCAL PETROV-GALERKIN (MLPG) UTILIZANDO PROCESSADORES GRÁFICOS (GPU) E CUDA

Bruno Carvalho Correa

DATA DA DEFESA: 24/02/2014

Universidade Federal de Minas Gerais
Escola de Engenharia
Programa de Pós-Graduação em Engenharia Elétrica

Paralelização do método Meshless Local Petrov-Galerkin (MLPG) utilizando processadores gráficos (GPU) e CUDA

Bruno Carvalho Correa

Dissertação de mestrado submetida à Banca Examinadora designada pelo Colegiado do Programa de Pós-Graduação em Engenharia Elétrica da Universidade Federal de Minas Gerais, como requisito parcial para obtenção do título de Mestre em Engenharia Elétrica.

Orientador: Prof. Renato Cardoso Mesquita

Belo Horizonte, novembro de 2013

Resumo

Neste trabalho desenvolvem-se estratégias de paralelismo para o método computacional sem malha Petrov-Galerkin local (MLPG) a ser executado em uma arquitetura altamente paralela conhecida como unidade de processamento gráfico (GPU). Métodos sem malha vem ganhando cada vez mais destaque dentre os métodos computacionais para se resolver problemas regidos por equações diferenciais parciais. Ao contrário do consagrado método dos elementos finitos (FEM), este não precisa de uma malha para obtenção da matriz global do sistema. O domínio do problema é representado por uma distribuição de nós mais uma descrição da fronteira, que na verdade nada mais é do que nós também distribuídos ao longo das bordas do problema, além das condições de contorno. Com a intenção de se obter melhor desempenho do algoritmo, neste trabalho aplica-se o mesmo em uma arquitetura altamente paralela. Diversas aplicações vem sendo desenvolvidas para tal arquitetura, em especial pode-se destacar o modelo de programação CUDA, do inglês computer unified architecture. Esta arquitetura tem mostrado uma imensa versatilidade de forma que é possível delegar computações numéricas utilizando a GPU como um coprocessador auxiliando a unidade central de processamento (CPU). O método sem malha MLPG é paralelizado de forma a ganhar tempo e precisão já que é possível aumentar o número de nós no domínio até um limite bem maior que se fosse executado na CPU. Para testar o algoritmo paralelizado, ele é aplicado a um problema eletromagnético clássico que possui solução analítica, e o tempo de execução na GPU é comparado com o tempo obtido pelo mesmo algoritmo executado na CPU. Os resultados obtidos com a GPU GeForce GTX 680 da NVIDIA mostram que é possível se obter um tempo de execução até 20 vezes menor com o algoritmo paralelo, mantendo-se a mesma precisão da solução.

Abstract

In this work, a new strategy to paralelize the Meshless Local Petrov-Galerkin method(MLPG) is developed. It is executed in a high parallel architecture, the well known graphics processing unit (GPU). The meshless methods are extensively applied nowadays to solve several different problems of partial differential equations. Compared with the traditional finite element methods, the meshless methods are a quite interesting alternative because they do not require a mesh in order to solve a physical problem, only a node distribution and a proper description of the boundary of the problem (that is actually a node distribution on the boundary) as well as the boundary conditions are needed. In this work the algorithm is adapted to run on the GPU. Several applications are being developed to execute in this new architecture to take advantage of its high parallel nature. Among several models of programming, one can distinguish CUDA or Computer Unified Architecture of NVIDIA. CUDA is a scalable parallel architecture developed by NVIDIA and can be programmed in C or via graphics API, so that the GPU can be used as a coprocessor auxilliating the central processing unit (CPU) as well as serving as a cheap supercomputer for numerical applications with surprisingly readiness. The MLPG is parallelized to execute completly on the GPU side. The MLPG was chosen because of it's simplicity and because it does not require any complex geometric representation of the domain or any sychronization scheme to obtain the global system of equations. In order to test this approach, it is applied to an electromagnetic problem whose analytical solution exist. The execution time of both GPU and CPU versions are compared. The results obtained with NIVIDA GeForce GTX 680 in this work shows that it is possible to have an execution time 20 times smaller than the counterpart algorithm on the CPU, ensuring the same precision of results.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Símbolos

α, α_u	parâmetro do método da penalidade
α_s	parâmetro adimensional para obtenção do domínio de suporte
ϵ	permissividade elétrica
Γ	fronteira do domínio do problema
Γ	fronteira do problema
Γ_q	fronteira de Neumann
Γ_u	fronteira de Dirichlet
Γ_{sq}	intercessão do subdomínio com a fronteira de Neumann
Γ_{su}	intercessão do subdomínio com a fronteira de Dirichlet
$\hat{n}$	vetor normal
$\hat{u}_j$	nós fictícios utilizados pelo MLS para aproximar uma função
μ	permeabilidade magnética
Ω	domínio do problema
Ω_s	subdomínio local, onde avalia-se a forma fraca
ϕ_j	funções de forma
ρ	densidade de carga elétrica
σ	condutividade elétrica
$\tilde{q}$	valor da variável secundária do problema físico em questão

$\tilde{u}$	valor da função na fronteira de Dirichlet
$\vec{A}$	vetor potencial magnético
$\vec{B}$	vetor densidade de fluxo magnético
$\vec{D}$	vetor densidade de fluxo elétrico
$\vec{E}$	vetor campo elétrico
$\vec{H}$	vetor campo magnético
$\vec{J}$	vetor densidade de corrente elétrica
d_c	tamanho característico para se determinar os nó no domínio de suporte
d_s	distância entre nós
F_j	elemento j do vetor F
k	número de onda
K_{ij}	elemento (i, j) da matriz K
R_0	raio de atuação da função de teste
U_s	vetor de valores da função aproximada no MLPG

Capítulo 1

Introdução

Na engenharia existem diversos problemas complexos que necessitam de uma análise criteriosa e de alta fidelidade. Podem-se citar vários exemplos de sistemas complexos, dentre eles estão: a construção de estruturas metálicas, ou utilizando-se compósitos e até materiais mistos; problemas de cálculo de dinâmica dos fluidos, com o objetivo de minimizar o peso de uma aeronave por exemplo; cálculo do campo eletromagnético gerado por uma antena com determinada geometria, para se otimizar o consumo de energia e melhorar a fidelidade do sinal na maior área de cobertura possível; projeto e otimização de um conjunto de antenas. Todos esses sistemas são projetados para servir em uma larga faixa de condições de operação, mantendo o custo o menor possível. Atender todas essas condições requer soluções de compromisso entre requisitos conflitantes, para isso é necessária uma análise computacional precisa, o que exige cada vez mais soluções inovadoras.

1.1 Motivação

O método dos elementos finitos (do inglês Finite Element Method - FEM) tem sido utilizado largamente na indústria automobilítica, marinha, aeronáutica, aeroespacial, exploração de óleo e gás, dentre outras, para se resolver problemas diversos, devido à sua flexibilidade e versatilidade. Porém o FEM requer a construção de uma malha sobre o domínio do problema, que pode ser uma distribuição de segmentos de reta no caso de domínios unidimensionais, triângulos no caso bidimensional ou tetraedros no caso tridimensional. A construção dessa malha é uma tarefa cara do ponto de vista computacional, e muitas vezes requer intervenção humana quando se trata de geometrias tridimensionais mais complexas. Além disso é necessário gerar novamente a malha em problemas com fronteiras móveis ou com grandes deformações. Para

isso os métodos sem malha são uma alternativa interessante pois eliminam a dificuldade de gerar a malha várias vezes sobre o domínio do problema. No métodos sem malha, nós são simplesmente adicionados ou removidos. Além disso esses métodos eliminam problemas como a distorção de elementos, existentes no FEM [7]. A figura a 1.1 mostra a diferença entre os métodos sem malha e o FEM, nos métodos sem malha não há nenhuma conectividade entre os nós distribuídos pelo domínio do problema.

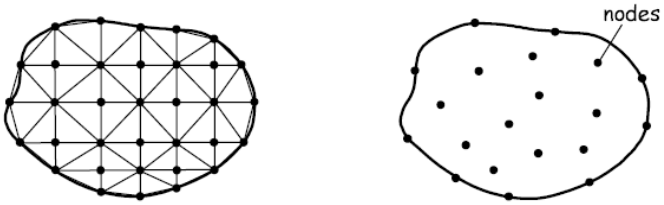


Figura 1.1: Modelagem utilizando o FEM à esquerda e utilizando Métodos sem malha à direita. Figura retirada de [14]

Diversos métodos sem malha foram desenvolvidos com o objetivo de se contornar as limitações do FEM, dentre eles pode-se destacar o método Element Free Galerkin (EFG) [18]. Apesar de não ser necessários uma malha para gerar as funções de forma e de teste, no método EFG é necessária a criação de um grid de fundo para realizar a integração da forma fraca. O método sem malha Petrov-Galerkin Local (Meshless Local Petrov-Galerkin - MLPG) [7], em contraste, se caracteriza por não necessitar de uma malha nem para a geração das funções de forma e de teste, nem para a integração da forma fraca.

O método MLPG é escolhido neste trabalho pela sua simplicidade na obtenção da solução, ou para ser mais preciso, pelo fato de não precisar de nenhuma estrutura complexa para representação do domínio. Nele é necessário somente conhecer a localização espacial dos nós e seus vizinhos, assim como uma descrição das fronteiras do problema. Além disso, o MLPG propicia uma formulação já paralelizada, ou seja, o método é baseado numa forma fraca local da equação diferencial governante, que uma vez obtida deve ser avaliada em um subdomínio local de cada nó. Em essência, cada nó no domínio do problema deve computar a integração das funções de forma e de teste, e para isso é necessário determinar somente os nós vizinhos que influenciam cada subdomínio. No MLPG cada subdomínio contribui para a construção de uma linha da matriz global do sistema. Portanto, o método já é paralelo por sua própria natureza. Essa localidade da computação e do acesso à memória é o que faz do MLPG um método extremamente vantajoso de se executar numa arquitetura de processador

massivamente paralela. Além disso, ele requer poucos nós no seu domínio de suporte local, o que implica em menos acesso à memória.

Os processadores gráficos modernos (graphics processing unit - GPU) evoluíram de um processador gráfico com pipeline de função fixa, para um processador gráfico paralelo programável, durante a última década. Esse foi um momento importante para a computação numérica, pois abriu novas oportunidades a serem exploradas. Anteriormente, o tempo de execução de um determinado código podia ser determinado simplesmente contando-se quantas operações de ponto flutuantes eram realizadas. Mais recentemente, a limitação da largura (taxa de bytes por segundo) de acesso à memória e latência de busca, tem sido um dos fatores dominantes no tempo gasto para um código executar, o que fez com que os fabricante de processadores (CPU - central processing unit) dedicassem grande área do processador para mitigar esse efeito. Na CPU a área dedicada para cache, predição e especulação é considerável se compararmos com a área do chip ocupada por unidades funcionais. No entanto, este cenário está mudando e a GPU se tornou recentemente um conjunto de unidades de processamento programáveis, substituindo a área dedicada para cache de memória e combatendo a latência de acesso com o paralelismo massivo. A figura 1.2 a seguir ilustra, de maneira abstrata, essa diferença de tamanho de área no chip dedicada a processamento de dados (de unidades lógica aritméticas - ALUs) na CPU e na GPU. Da figura, pode-se concluir que a GPU dedica maior área no chip para processamento de instruções.

Figura 1.2: A GPU dedica maior área do chip para processamento de dados. Figura retirada de [5]

Em novembro de 2006, a arquitetura NVIDIA Tesla foi introduzida pela primeira vez com a GeForce 8800 [8], quando o pipeline foi unificado tornando possível que aplicações paralelas de alto desempenho fossem escritas em linguagem C, utilizando o seu padrão de computação unificada ou CUDA (do inglês computer unified architecture). Neste trabalho é explorada uma arquitetura sucessora da NVIDIA Tesla, a arquitetura Kepler. Na figura 1.3 é apresentado um comparativo entre a quantidade de operações de ponto flutuantes realizadas por várias GPUs CPUs ao longo do anos. Nota-se que a quantidade de GFLOPs (do inglês *giga floating point*

operation per second, equivalente a 10^9 operações de ponto flutuante por segundo) das GPUs ultrapassa aproximadamente em seis vezes a velocidade das operações de ponto flutuante das CPUs atuais.

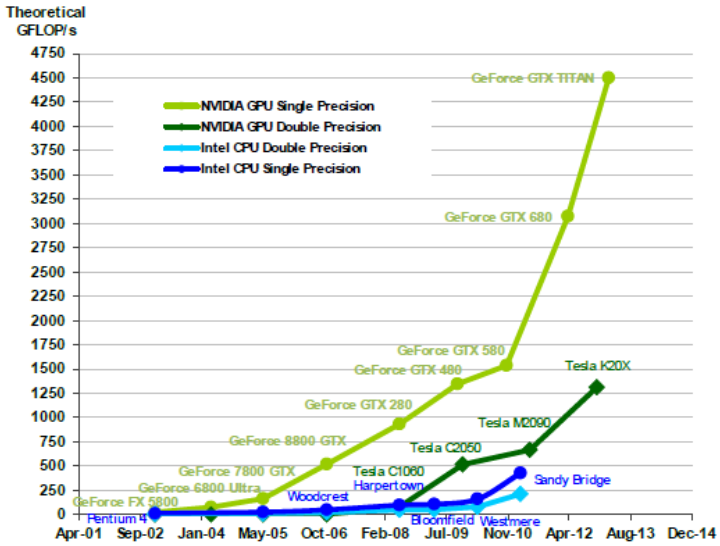


Figura 1.3: Comparação em GFLOPs entre CPUs e GPUs atuais. Gráfico retirado de [5]

CUDA é um modelo de programação como denominado pela NVIDIA, ou para ser mais preciso é uma pequena extensão das linguagens de programação C e C++. Ao escrever programas em CUDA o programador divide o programa em uma parte a ser executada de maneira serial e outra em *kernels* ou núcleos paralelos. A parte serial é executada na CPU enquanto a parte paralela ou *kernels* é executada na forma de um conjunto de *threads* na GPU. O programador deve organizar as *threads* em uma hierarquia de blocos e *grids*, como será descrito em capítulos posteriores. O modelo de programação CUDA é similar ao modelo "programa único múltiplos dados"(do inglês single-program multiple data - SPMD) [8]. Esta hierarquia de *threads* é escalável para qualquer GPU, independente se a mesma tem dezenas de processadores ou milhares.

1.2 Objetivo

A partir de um estudo do método MLPG, serão apresentadas duas formas alternativas do método que serão paralelizadas utilizando CUDA. Essas duas alternativas são obtidas escolhendo diferentes métodos para geração das funções de forma. Ambas as abordagens são comparadas quanto ao tempo de execução e quanto à precisão da solução em relação à solução analítica. Também é analisada a convergência da solução à medida em que se aumenta o número de nós

no domínio. O mesmo algoritmo presente na GPU será adaptado para executar na CPU, para se comparar o tempo de execução e também quantificar o ganho real da abordagem paralela em relação à serial. Portanto, o objetivo geral do trabalho é conseguir um ganho significativo no tempo de execução com o algoritmo na GPU, e discutir qual das duas abordagens do MLPG melhor se adapta à arquitetura paralela da GPU, em função dos resultados obtidos.

Capítulo 2

Conceitos Básicos

2.1 Métodos sem malha

Métodos sem malha diferem do FEM fundamentalmente pelo fato de que no primeiro as funções de forma são construídas durante a análise, e dependem da localidade do ponto de interesse sendo única para o mesmo, e no FEM as funções de forma são idênticas para elementos idênticos além do fato de serem predeterminadas para diferentes tipos de elementos, antes mesmo da análise começar. Além disso, existem outros detalhes como o fato das funções de forma serem mais complexas e, quando se utiliza integração numérica, ser necessário um maior número de pontos de integração para se obter a precisão adequada na solução nos métodos sem malha.

Outro ponto é a discretização do domínio do problema. No FEM o domínio do problema é definido pelas curvas ou superfícies dos elementos. Se retas são utilizadas para representar os elementos, como geralmente é utilizado em situações práticas, as fronteiras do domínio são representadas por retas ou planos [13]. A exatidão da solução depende então da quantidade de elementos e da ordem dos mesmos. Uma malha mais fina gera uma melhor aproximação da solução, porém requer maior quantidade de recursos computacionais. A figura 2.1 mostra um exemplo sobre como o domínio do problema é discretizado no FEM.

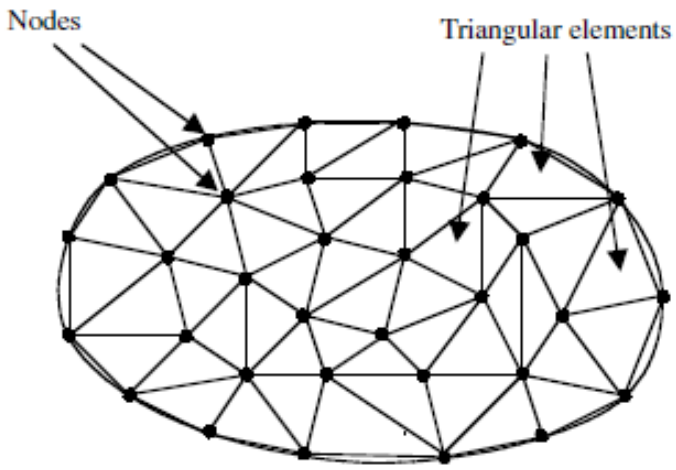


Figura 2.1: Domínio do problema discretizado no FEM. Figura retirada de [13]

Nos métodos sem malha o domínio é representado pelos nós. Entre quaisquer dois pontos, o valor deve ser interpolado utilizando as funções de forma. Entretanto, é necessário discretizar a fronteira de alguma forma para representar a intersecção do subdomínio local a cada nó com o domínio global. Abaixo segue a figura 2.2 que mostra como o domínio é aproximado nos métodos sem malha.

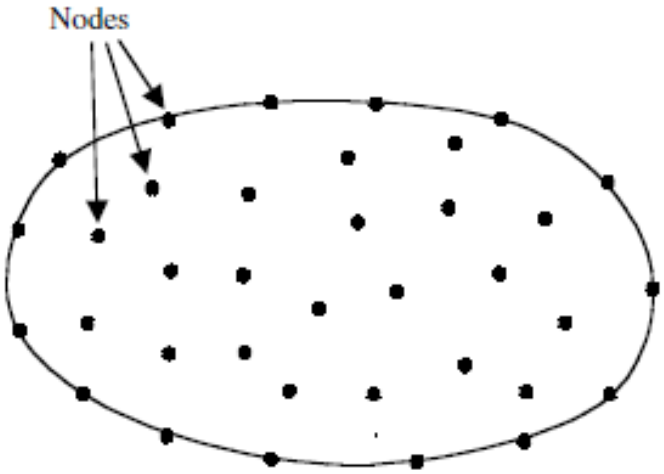


Figura 2.2: Domínio do problema discretizado nos métodos sem malha. Figura retirada de [13]

Adicionalmente, no FEM as funções de forma são pré-determinadas, têm expressão analítica conhecida, e por isto a computação das funções de forma é direta. Além disso, todas as funções de forma nos elementos finitos obedecem a propriedade da função de delta de Kronecker. Nos métodos sem malha as funções de forma não têm expressão analítica e, usualmente, é utilizado o método dos mínimos quadrados móveis (MLS) para determiná-las. Porém, as funções de forma geradas pelo MLS não possuem a propriedade da função delta de Kronecker, e se faz necessário o uso de técnicas adicionais para se impor as condições de contorno. Além do MLS existe outro método utilizado para construção das funções de forma que também é utilizado neste trabalho. O PIM, do inglês *Point Interpolation Method*, produz funções de forma que possuem a propriedade da função delta de Kronecker, o que torna a imposição das condições de contorno essenciais direta (sem uso de técnicas adicionais).

Ao se resolver um problema físico como, por exemplo, um problema eletrostático ou de propagação de ondas eletromagnéticas, deve-se obter primeiro a equação diferencial parcial que descreve o fenômeno. No caso do eletromagnetismo, o ponto de partida são as equações de Maxwell que regem todos os fenômenos elétricos em escala macroscópica. Abaixo seguem as equações de Maxwell na forma diferencial:

$$\nabla \times H = J + \frac{\partial D}{\partial t}, \tag{2.1a}$$

$$\nabla \times E = -\frac{\partial B}{\partial t}, \tag{2.1b}$$

$$\nabla \cdot D = \rho, \tag{2.1c}$$

$$\nabla \cdot B = 0, \tag{2.1d}$$

Uma vez estabelecida a equação diferencial governante, ela pode ser discretizada ou reescrita numa forma equivalente. No FEM, um conjunto de equações discretas pode ser obtido a partir da aplicação de um método variacional, ou de um de resíduos ponderados. No caso do método dos elementos finitos tem-se uma forma fraca, enquanto as outras duas abordagens partem da equação diferencial governante.

Após a escolha das funções de teste e de forma em cada elemento (no caso de métodos com malha) ou subdomínio (para o caso sem malha), um subsistema é produzido e contribui para uma parcela da matriz do sistema global. Portanto, após feitos todos os cálculos é montado

um sistema matricial de equações que deve ser submetido a um método computacional para resolvê-lo. Em geral existem duas categorias de métodos para solucionar o sistema de equações, métodos diretos e métodos iterativos. Os primeiros se caracterizam pela decomposição da matriz de equações e necessitam do sistema completo disponível na memória e, no caso de sistemas matriciais esparsos, geram novos elementos onde antes existiam zeros na matriz.

Como métodos iterativos pode-se citar o método de gradientes conjugados, de gradientes biconjugados e o método do mínimo resíduo generalizado (GMRES), dentre outros. Estes são mais indicados à sistemas esparsos muito grandes pois não geram coeficientes não nulos durante o processo de solução. Também são particularmente interessantes para algoritmos paralelizados, ou seja, quando se tem a matriz do sistema parcialmente disponível, e pretende-se resolver o sistema sem tê-lo montado por completo. Esta situação ocorre no FEM ou nos métodos sem malha, onde cada linha de processamento ou *thread* produz uma porção da matriz global.

Outra característica dos métodos sem malha é que eles demandam um maior uso da CPU para a montagem da matriz global, uma vez que, as funções de forma são calculadas durante a montagem do sistema. Essa é uma tarefa que deve ser calculada para cada subdomínio ou elemento, por isso é melhor se feita em paralelo. Para métodos sem malha baseados no MLS, existe ainda um esforço computacional extra pelo fato deste precisar de técnicas adicionais para forçar as condições de contorno essenciais.

2.2 Procedimento geral do MLPG

A figura 2.3 mostra de maneira geral os passos necessários para se implementar o MLPG em comparação com o método dos elementos finitos. Muito do que está descrito no diagrama já foi discutido na seção anterior. Nesta seção pretende-se detalhar mais o MLPG, as particularidades da construção das funções de forma e suas propriedades, a imposição das condições de contorno e discorrer sobre a definição do domínio de suporte e domínio de influência.

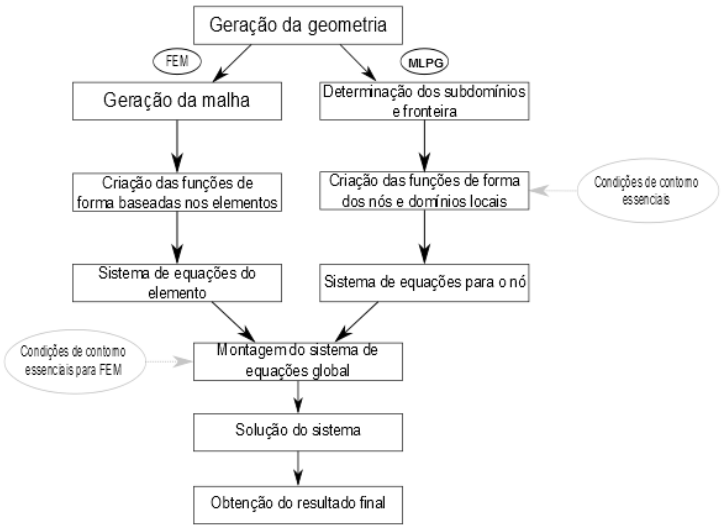


Figura 2.3: Esquema comparativo do método MLPG com o FEM.

Um passo importante da solução numérica de um problema em eletromagnetismo é a obtenção de uma boa discretização do domínio. Nos métodos sem malha, como já foi dito, essa discretização é obtida a partir de uma distribuição de nós em todo domínio, incluindo-se, também, a fronteira do problema, onde são impostas as condições de contorno.

A distribuição de nós pode ser uniforme, mas geralmente esta não é uniforme, como ilustrado na figura 2.4, e prefere-se uma maior densidade de nós onde o gradiente da função que se está aproximando é mais intenso. Alguns algoritmos podem ser utilizados para se obter a distribuição de nós de maneira adaptativa no próprio código dos métodos sem malha.

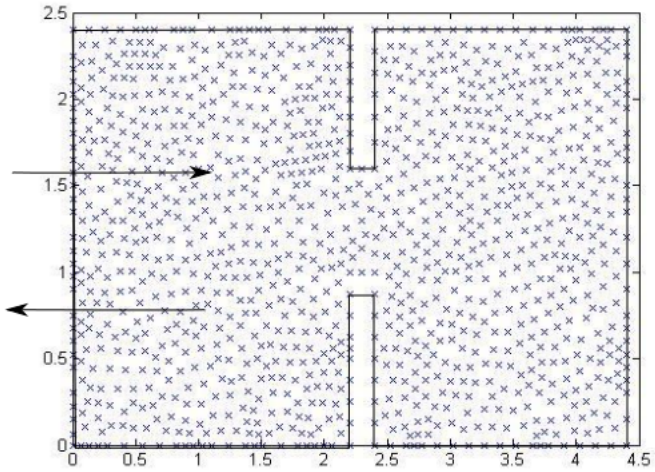


Figura 2.4: Exemplo de uma distribuição de nós não uniforme num domínio que representa uma cavidade eletromagnética retangular.

A segunda etapa do método é interpolar o valor da variável a ser calculada $u(x, y, z)$ (que pode ser, por exemplo, o potencial escalar campo elétrico ou magnético para problemas de eletromagnetismo) em cada ponto no domínio do problema. Como não existe uma malha de elementos a variável deve ser interpolada considerando os nós que formam o domínio de suporte do ponto $\mathbf{x} = (x,y,z)$.

$$u(\mathbf{x}) = \sum_{i=1}^{n} \phi_i(\mathbf{x}) u_i = \Phi(\mathbf{x}) U_s \tag{2.2}$$

Onde *n* é o número de nós no domínio de suporte do ponto **x**, u_i é a variável nodal para i-ésimo nó,U_s é o vetor que representa todos os valores da variável nodal em todos os nós do subdomínio, e $\phi_i(x)$ são as funções de forma calculadas utilizando-se os nós do domínio de suporte de **x**. O domínio de suporte de um ponto **x** determina os nós utilizados para dar suporte ou aproximar o valor de uma função no ponto **x**.

Existem duas formas de se encontrar os nós que compõem o domínio de suporte de um ponto. Na primeira, o domínio de suporte é determinado a partir do conceito de domínio de influência. Cada nó definido no problema possui um domínio de influência com um certo raio, ao se aproximar a função de forma em um ponto **x** qualquer deve-se primeiro determinar quais

são os nós cujos domínios de influência incluem o ponto **x** em questão, como ilustrado na figura 2.5.

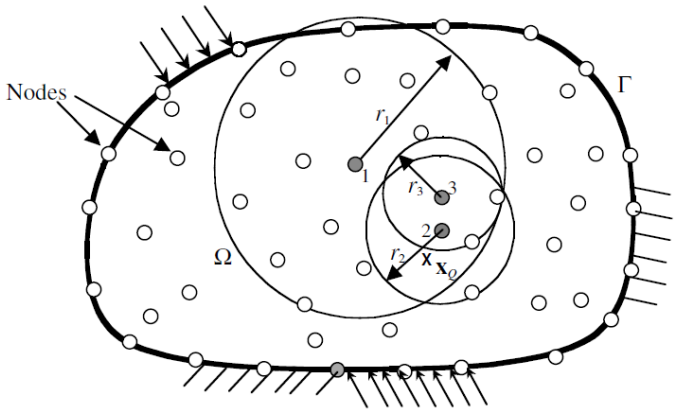


Figura 2.5: Domínios de influência. Ao construir a função de forma para o ponto **x** em $\mathbf{x_Q}$, os nós cujos domínios de influência incluem o ponto x_Q são levados em consideração. Como, por exemplo, os nós 1 e 2 são considerados, porém o nó 3 não. Figura retirada de [13].

Este domínio de influência pode ainda ser determinado por uma função peso em função da distância do ponto ao nó central, caindo à zero quando o ponto está fora do raio de influência do nó, como está ilustrado na figura 2.6. Essa característica garante que os nós entram e saem de maneira suave à medida em que se move o ponto **x** pelo domínio.

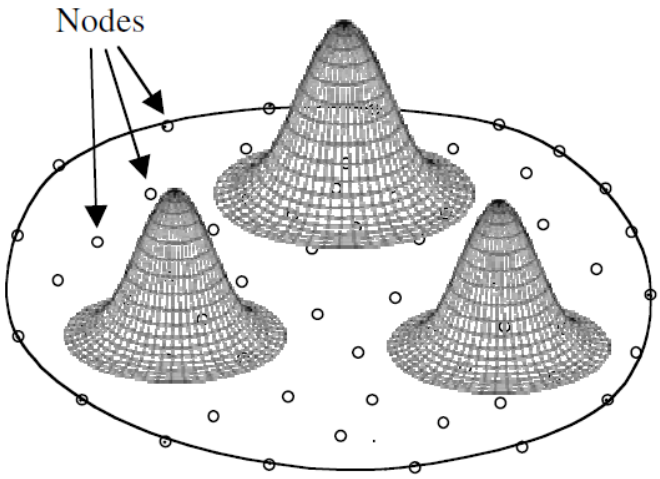


Figura 2.6: Domínio de influência dos nós definidos por uma função peso suave, cujo valor cai a zero para pontos fora do raio de influência. Figura retirada de [13]

Na segunda forma, o domínio de suporte de um ponto **x** é determinado traçando-se um raio ao redor do mesmo, de forma que todos os nós incluídos nesta região delimitada são escolhidos para contribuir para a aproximação da função no ponto. Esta região pode ser retangular ou circular como ilustrado na figura 2.7.

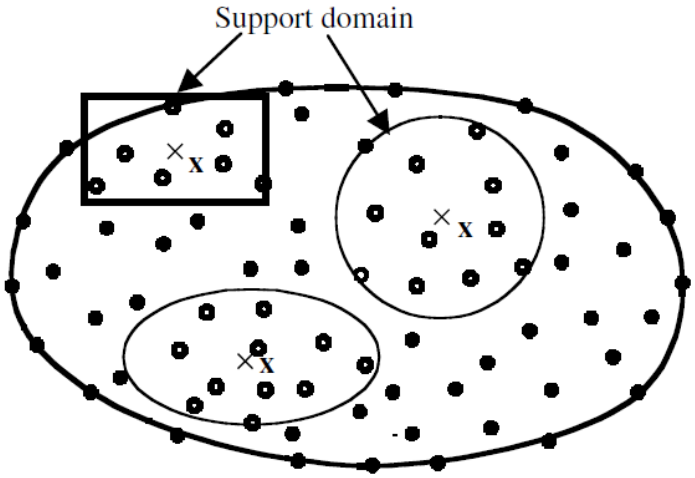


Figura 2.7: O domínio de suporte determina os nós utilizados para aproximar o valor de uma função em um determinado ponto $\mathbf{x}$. Nesta figura ilustra-se uma das maneiras de se obter os nós no domínio de suporte.

Como os domínio de influência podem ser diferentes de nó para nó, implicando que um nó pode ter um maior raio de influência do que outro, o não balanceamento das funções de forma, decorrentes da má distribuição de nós pelo domínio, deve ser evitado.

Segundo [13] a qualidade da interpolação depende diretamente do número de nós em cada domínio de suporte do ponto. Para este método de seleção do domínio de suporte, a escolha do raio determina o número de nós selecionados. Este número pode ser determinado pela seguinte equação 2.3.

$$d_s = \alpha_s d_c \tag{2.3}$$

Onde α_s é um parâmetro adimensional e d_c é o tamanho característico, que relaciona o espaçamento dos nós ao redor do ponto de interesse. Em essência, os nós dentro de um

raio d_s do ponto em questão fazem parte do seu domínio de suporte. Se a distribuição é uniforme, então d_c é simplesmente a distância entre dois nós vizinhos.

Geralmente o valor de α_s é dependente do problema físico, porém neste trabalho adota-se o tamanho de α_s como sendo aquele raio que inclui o mínimo necessário de nós cujo valor seja possível realizar a interpolação local, o que será discutido posteriormente.

Uma próxima etapa do processo dos métodos sem malha é a imposição das condições de contorno, que é realizada durante a construção das funções de forma. Porém se as funções de forma utilizadas possuírem a propriedade de delta de Kronecker, esta etapa pode ser adiada para a etapa de montagem do sistema global. A propriedade da função de delta de Kronecker, pode ser definida como:

$$\phi_i(\mathbf{x}_j) = \begin{cases} 1, & \text{i} = \text{j} \\ 0, & \text{i} \neq \text{j; i,j} = 1,2,3,...,n \end{cases} \tag{2.4}$$

Essa condição permite a imposição direta das condições de contorno essenciais. As funções de forma obtidas pelo RPIM possuem essa propriedade, enquanto as obtidas pelo MLS não a possuem.

A última etapa do método envolve a solução do sistema de equações. No caso do MLPG, que é objeto de estudo neste trabalho, a matriz global do sistema é uma matriz definida por bandas e muito esparsa (muitos elementos nulos), assim como no FEM. Porém no MLPG há uma dificuldade com o mal condicionamento do sistema, o que influencia diretamente na escolha no método de solução do sistema.

Capítulo 3

Meshless Local Petrov-Galerkin - MLPG

O método sem malha Petrov-Galerkin local foi desenvolvido e utilizado pela primeira vez por Zhu, Zhang and Cho (1998) para resolver problemas lineares e não-lineares. O MLPG se caracteriza como um verdadeiro método sem malha pois não precisa de uma malha, nem para interpolação das funções de forma e de teste, nem mesmo para realizar a integração da forma fraca. Todas as integrais podem ser realizadas através de subdomínios com formatos regulares e suas fronteiras.

Primeiramente é obtida uma forma fraca local a partir da forma peso-residual da equação diferencial governante **??**. Neste trabalho se explora dois métodos para aproximação das funções de forma, o método dos mínimos quadrados móveis (MLS), e o método RPIMp (do inglês *Radial Point Interpolation method*) que utiliza funções de base radial com polinômio de primeira ordem (RPIMp). Após determinada a função de forma, são escolhidas as funções de teste. No método MLPG as funções de forma e de teste podem ser diferentes. As funções de teste utilizadas neste trabalho são a função de Heaviside, como definido em Atluri e Shen [7]. Estas funções de teste são escolhidas principalmente para se evitar uma integração no subdomínio, como será mostrado mais adiante. As condições de contorno essenciais são impostas diretamente quando se utiliza as funções de base radial (do inglês *radial basis function* - RBF) como função de forma, e através do método da penalidade quando se utiliza o método dos mínimos quadrados móveis. Finalmente, a substituição das funções de forma e de teste na forma local fraca da equação diferencial governante resulta em um sistema de equações

algébricas, e a solução deste sistema é obtida através de um método direto ou iterativo, como mencionado anteriormente.

3.1 Formulação unidimensional do MLPG

Considerando a equação diferencial unidimensional abaixo:

$$\frac{d}{dx}(b(x)\frac{du}{dx}) + c(x)u = f(x) \tag{3.1}$$

No domínio Ω $(0 \leq x \leq l)$, com fronteira Γ, onde as funções b(x) e c(x) são parâmetros do problema, e f(x) é alguma fonte de carga também em função de x. Submetido às condições de contorno.

$$u = \tilde{u} \text{ em } \Gamma_u, q = \tilde{q} \text{ em } \Gamma_q \tag{3.2}$$

Onde

$$q = b\frac{du}{dx} \tag{3.3}$$

$\Gamma_u = x(0)$ e $\Gamma_q = x(l)$ são as fronteira onde u e q são descritas pelas condições de contorno. As variáveis u e q representam grandezas físicas diferentes. No caso de problemas de transferência de calor, u é a temperatura, b é a condutividade térmica, f é a geração de calor, e $q = b\frac{du}{dx}$ é o fluxo de calor.

Para se obter uma aproximação da solução da equação diferencial acima, se aplica uma técnica de peso residual. Portanto, como se deseja uma aproximação para *u*, o erro dessa aproximação em cada ponto gera um resíduo dado por:

$$R = -\frac{d}{dx}(b\frac{du}{dx}) + cu - f \tag{3.4}$$

Multiplicando-se esse resíduo por uma função peso $v(x)$, integrando o produto sobre todo o domínio do problema e fazendo-se o resultado dessa integral igual a zero obtém-se:

$$\int_\Omega v \left[-\frac{d}{dx}\left(b\frac{du}{dx} \right) + cu - f \right]_\Gamma dx = 0 \tag{3.5}$$

Onde Γ denota a fronteira do problema. Essa forma global requer que a solução $u(x)$ seja C^1 contínua, o que significa que a solução deve ser duas vezes diferenciável. Esta é uma restrição não muito difícil para problemas unidimensionais, porém para problemas de dimensões maiores é uma restrição difícil de satisfazer.

Entretanto, aplicando-se integração por partes sobre a derivada segunda $\frac{d}{dx}(b\frac{du}{dx})$ se obtém:

$$\int_\Omega b\frac{du}{dx}\frac{dv}{dx}dx + \int_\Omega cvudx - \int_\Omega fvdx - [vb\frac{du}{dx}]_{x=0}^{x=l} = 0 \tag{3.6}$$

As restrições sobre $u(x)$ foram enfraquecidas pois a forma acima só requer que *u* seja C^0 contínua, derivável uma única vez. Da mesma forma para *v*, *u* deve satisfazer agora somente as condições essenciais de contorno (ou condições de Dirichlet), uma vez que, as condições naturais de contorno já estão presentes na forma local fraca. Essa equação acima então é chamada de forma fraca da equação diferencial governante.

A variável secundária, que neste caso unidimensional é $q = b\frac{du}{dx}$, é descontínua no FEM nas fronteiras dos elementos [14]. Isso se deve ao fato das funções de forma serem definidas linearmente por partes. No MLPG a aproximação da função também é feita por combinação linear de funções de forma:

$$u(x) = \sum_{j=1}^{n} \hat{u}_j \phi_j \tag{3.7}$$

Nesta equação *n* é o número de nós do domínio de suporte do ponto onde a aproximação está sendo efetuada, $\hat{u}_j$ são os valores nodais de aproximação nos nós e ϕ_j são as funções de forma.

Além do problema da descontinuidade das variáveis secundárias, pretende-se também eliminar a necessidade de uma malha de fundo para a integração dos termos da forma fraca, de maneira a tornar um método verdadeiramente sem malha. Para isso, foi proposto por [16] a escolha da função de teste *v* de um espaço diferente da função de forma:

$$v_i \neq \phi_i \tag{3.8}$$

Assim, qualquer função de teste não nula pode ser escolhida.

Em [7] são sugeridas e testadas seis funções de teste, resultando em diferentes variações do MLPG, como esquematizado na tabela 3.1:

Tabela 3.1: Variações do método MLPG. Tabela retirada de [7].

Método	Função de teste	Integral para avaliar a forma fraca
MLPG1	Função peso do MLS	integral de domínio
MLPG2	Delta de Dirac $\delta(\mathbf{x},x_I)$	Nenhuma
MLPG3	Mínimos quadrados $\phi_{,ii}(x)$	Integral de domínio
MLPG4	Solução fundamental	Integral de fronteira singular
MLPG5	Função degrau Heaviside	Integral de fronteira regular
MLPG6	Mesma da função de forma	Integral de domínio

Dentre todas as seis versões do MLPG na tabela acima, o MLPG5 é particularmente interessante, pois elimina a necessidade de avaliar o termo da derivada da função de teste na forma fraca, uma vez que essa função é constante e de valor unitário dentro do seu domínio.

As funções de teste no MLPG têm a característica de se anular dentro de uma certa distância. Como ilustrado na figura 3.1, a função de teste para o nó *i* tem um raio de atuação R_o. Essas

características locais que as funções de teste e as funções de forma possuem é o que dão ao método sua característica local, como pode ser visto na figura 3.1.

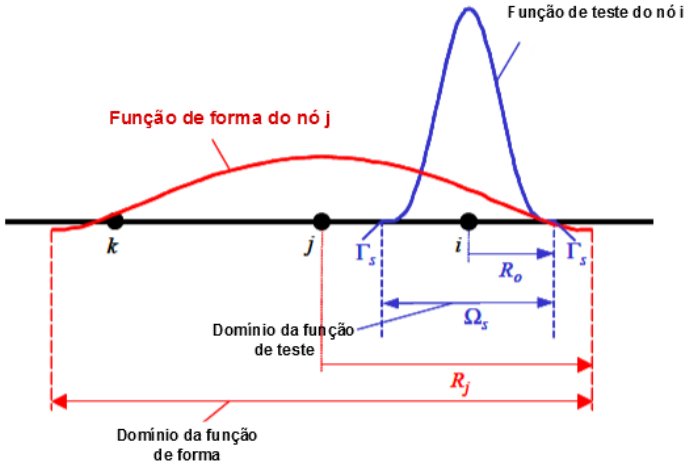


Figura 3.1: Domínio da função de teste e de forma. Figura adaptada de [14]

A possibilidade de escolha das funções de teste e de forma em espaços diferentes é o que caracteriza este como um método Petrov-Galerkin. Como a função de teste tem valores diferentes de zero somente dentro do domínio Ω_s, pode-se reescrever a forma fraca da equação 3.6:

$$\int_{\Omega_s} b\frac{du}{dx}\frac{dv}{dx}dx + \int_{\Omega_s} cvudx - \int_{\Omega_s} fvdx - [vb\frac{du}{dx}]_{\Gamma_s} = 0, \forall\Omega_s \tag{3.9}$$

Após obtida a forma fraca local (equação 3.9) deve-se garantir que as condições de contorno essenciais são obedecidas. Se as funções de forma satisfazem o delta de Kronecker isto é facilmente imposto. Se não, uma técnica adicional para forçar as condições de contorno deve ser usada. Neste trabalho utiliza-se o método da penalidade [16], que será detalhado mais adiante neste capítulo. A forma fraca local com penalidade se torna:

$$\int_{\Omega_s} b\frac{du}{dx}\frac{dv}{dx}dx + \int_{\Omega_s} cvudx - \int_{\Omega_s} fvdx - [vb\frac{du}{dx}]_{\Gamma_s} + \alpha_u[(u-\tilde{u})v]_{\Gamma_{su}} = 0, \forall\Omega_s \tag{3.10}$$

Onde α na equação acima é o parametro da penalidade, que geralmente é um valor muito alto em torno de 10^5 ou 10^6 dependendo da formulação do problema, Γ_{su} é a parte correspondente do subdomínio Ω_s que intercepta a fronteira Γ_u, ou a fronteira de Dirichlet.

Para finalizar o procedimento de obtenção da forma fraca, falta ressaltar que o termo $-[vq]_{\Gamma_s}$, que é avaliado na fronteira do subdomínio, é composto de três partes. A primeira denominada fronteira Γ_{su} que corresponde à parte do subdomínio que intercepta a fronteira de Dirichlet. A segunda Γ_{sq}, a parte que intercepta a fronteira de Neumann, ou fronteira Γ_q. E a Terceira representa a parte interna do subdomínio, ou parte que não intercepta nenhuma fronteira do problema, ou Γ_s. Pode-se escrever então:

$$\Gamma_{su} \equiv \Gamma_s \cap \Gamma_u, \Gamma_{sq} \equiv \Gamma_s \cap \Gamma_q, \Gamma_{si} \text{ (fronteira interna a } \Omega) \tag{3.11}$$

Reescrevendo a equação 3.10 obtém-se:

$$\begin{aligned} \int_{\Omega_s} b\frac{du}{dx}\frac{dv}{dx}dx + \int_{\Omega_s} cvudx - \int_{\Omega_s} fvdx - \left[vb\frac{du}{dx}\right]_{\Gamma_{su}} \\ - \left[vb\frac{du}{dx}\right]_{\Gamma_{sq}} - \left[vb\frac{du}{dx}\right]_{\Gamma_s} + \alpha_u \left[(u-\tilde{u})v\right]_{\Gamma_{su}} &= 0, \forall\Omega_s \end{aligned} \tag{3.12}$$

Onde a componente $-[vq]_{\Gamma_{si}}$ é nula pois v se anula na fronteira de Ω_s. Ou seja:

$$\int_{\Omega_s} b\frac{du}{dx}\frac{dv}{dx}dx + \int_{\Omega_s} cvudx - \int_{\Omega_s} fvdx - [vq]_{\Gamma_{sq}} - [v\tilde{q}]_{\Gamma_{sq}} + \alpha_u[(u-\tilde{u})v]_{\Gamma_{su}} = 0, \forall\Omega_s \tag{3.13}$$

As substituições das funções de forma e de teste geram uma sistema de equações lineares. Este sistema é não simétrico pois esta é uma característica dos métodos Petrov-Galerkin, pelo fato das funções de forma e de teste serem diferentes.

O sistema de equações produzido é:

$$\mathbf{K}.\hat{\mathbf{u}} = \mathbf{F} \tag{3.14}$$

Onde:

$$K_{ij} = \int_{\Omega_{si}} b\frac{d\phi_j}{dx}\frac{dv_i}{dx}dx + \int_{\Omega_s} cv_i\phi_j dx + \alpha_u[\phi_j v_i]_{\Gamma_{su}}, \tag{3.15a}$$

$$F_i = \int_{\Omega_{si}} fv_i dx + [v_i q]_{\Gamma_{su}} + [v_i\tilde{q}]_{\Gamma_{sq}} + \alpha_u[\tilde{u}v_i]_{\Gamma_{su}}, \tag{3.15b}$$

Neste sistema *i* corresponde a cada subdomínio Ω_s do problema e *j* a cada função de forma, ϕ_i. No MLPG5, o termo $\int_{\Omega_s} b\frac{d\phi_j}{dx}\frac{dv_i}{dx}dx$ na equação 3.15 se anula, dado que a função de Heaviside é constante no interior do subdomínio.

3.2 Formulação bidimensional do MLPG

Para a formulação bidimensional considera-se a equação de Poisson:

$$\nabla^2 u(x, y) = p \tag{3.16}$$

Onde *u* é a função potencial, e *p* é uma dada distribuição de densidade de carga. O resíduo da equação acima é então obtido e este é multiplicado por uma função peso, integrado e igualado a zero (como no caso unidimensional).

$$\int_{\Omega} w(\nabla^2 u(\mathbf{x}) - p)d\Omega = 0 \tag{3.17}$$

A diferença para a formulação unidimensional é que ao invés de se aplicar integração por partes para enfraquecer as restrições na solução, aplica-se o teorema da divergência onde:

$$\int_{\Omega} \nabla \cdot u(\mathbf{x}) d\Omega = \int_{\Gamma} u(\mathbf{x}) d\Gamma \tag{3.18}$$

Para isso aplica-se primeiro a seguinte identidade vetorial:

$$\nabla \cdot (\phi \vec{A}) = \vec{A} \cdot \nabla \phi + \phi \nabla \cdot \vec{A} \tag{3.19}$$

Onde ϕ é uma função escalar e $\vec{A}$ é uma função vetorial. Aplicando-se a identidade acima e o teorema da divergência, obtém-se a forma fraca em duas dimensões:

$$\int_{\Gamma} w \cdot \nabla u d\Gamma - \int_{\Omega} \nabla w \cdot \nabla u d\Omega - \int_{\Omega} w p d\Omega = 0 \tag{3.20}$$

A substituição das funções de teste resulta na forma fraca local, como visto na seção anterior. Por isso, pode-se substituir Ω por Ω_s e Γ por Γ_s, resultando na forma fraca local:

$$\int_{\Gamma_s} w \cdot \nabla u d\Gamma - \int_{\Omega_s} \nabla w \cdot \nabla u d\Omega - \int_{\Omega_s} w p d\Omega = 0, \forall \Omega_s \tag{3.21}$$

Na seção anterior se destacou também a utilização de diferentes funções de teste, resultando em variações do método MLPG, como pode ser visto na tabela 3.1. Nesta seção se utiliza a função de Heaviside como função de teste, ou seja, usa-se o MLPG5. Esta função é ilustrada na figura 3.2.

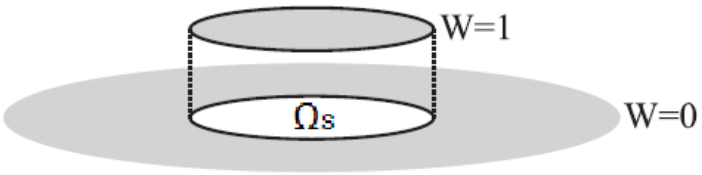


Figura 3.2: Função de Heaviside. Figura retirada de [9]

A função de Heaviside tem valor unitário dentro de um raio controlado que define o domínio de integração, denominado assim pois este é o domínio onde se realiza as integrações locais da forma fraca local.

$$W(\mathbf{x}) = \begin{cases} 1 & \mathbf{x} \in \Omega_s \\ 0 & \mathbf{x} \notin \Omega_s \end{cases} \tag{3.22}$$

A utilização da função de Heaviside elimina o segundo termo da equação 3.21, o que torna o método mais estável, pois nos métodos sem malha a integração é um problema, uma vez que o integrando não tem um comportamento polinomial como no FEM. Evitando-se uma integração no domínio o método fica menos susceptível a erros numéricos. Esta é a vantagem de se utilizar o MLPG5 como destacado em [7].

Com a substituição da função de Heaviside para função de teste, a equação 3.23 se torna:

$$\int_{\Gamma_s} \hat{n} \cdot \nabla u d\Gamma - \int_{\Omega_s} p d\Omega = 0, \forall \Omega_s \tag{3.23}$$

Onde $\hat{n}$ denota a componente normal a fronteira do subdomínio. Finalmente, a integração do subdomínio local é dividida entre integração na fronteira de Dirichlet, de Neumann e completamente interior ao domínio global como detalhado na figura abaixo.

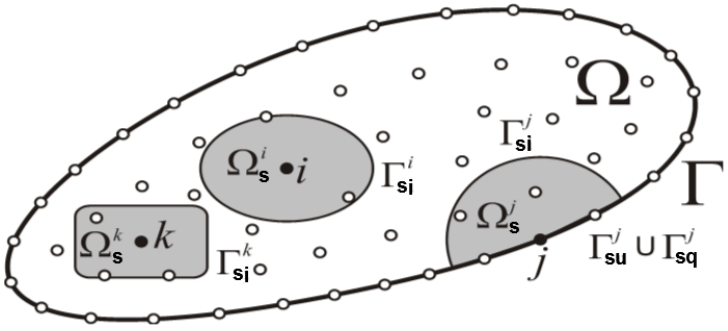


Figura 3.3: Domínio da função de teste e de forma. Phillips [14].

As fronteiras dos subdomínios são representadas da seguinte maneira:

$$\Gamma_{su} \equiv \Gamma_s \cap \Gamma_u, \Gamma_{sq} \equiv \Gamma_s \cap \Gamma_q, \Gamma_{si} \equiv \Gamma_s \text{ (fronteira interna a } \Omega) \tag{3.24}$$

Resultado na forma fraca local:

$$\int_{\Gamma_{su}} \hat{n} \cdot \nabla u d\Gamma + \int_{\Gamma_{sq}} \hat{n} \cdot \nabla u d\Gamma + \int_{\Gamma_{si}} \hat{n} \cdot \nabla u d\Gamma - \int_{\Omega_s} p d\Omega = 0, \forall \Omega_s \tag{3.25}$$

Acrescentando o termo do método da penalidade para impor as condições de contorno tem-se:

$$\int_{\Gamma_{su}} \hat{n} \cdot \nabla u d\Gamma + \int_{\Gamma_{sq}} \hat{n} \cdot \nabla u d\Gamma + \int_{\Gamma_{si}} \hat{n} \cdot \nabla u d\Gamma + \int_{\Gamma_{su}} \alpha(u - \tilde{u}) d\Gamma - \int_{\Omega_s} p d\Omega = 0, \forall \Omega_s \tag{3.26}$$

Para finalizar, as substituições das funções de teste e de forma na forma fraca local produzem o mesmo sistema de equações que no caso unidimensional, porém com os seguintes valores de **F** e **K**:

$$K_{ij} = \int_{\Gamma_{su}} \hat{n} \cdot \nabla \phi_j d\Gamma + \int_{\Gamma_{sq}} \hat{n} \cdot \nabla \phi_j d\Gamma + \int_{\Gamma_{si}} \hat{n} \cdot \nabla \phi_j d\Gamma + \int_{\Gamma_{su}} \alpha \phi_j d\Gamma \tag{3.27}$$

$$F_i = \int_{\Gamma_s ui} \alpha \tilde{u} d\Gamma + \int_{\Omega_s i} p d\Omega \tag{3.28}$$

Cada linha *i* da matriz K representa um subdomínio no problema (associado ao nó *i*), e cada coluna *j* está associada à função de forma de cada nó *j* que está no domínio de suporte do subdomínio *i*.

3.3 Construção das funções de forma

A escolha do método para criação das funções de forma é uma questão central do método MLPG. O método de aproximação das funções deve ser capaz de trabalhar com nós distribuídos

aleatoriamente sobre o domínio do problema, ou seja, uma distribuição que não segue um padrão bem definido. Além disso, não é definida nenhuma conectividade entre os nós.

Em [13] o autor destaca algumas características desejáveis para a construção das funções de forma para o MLPG, dentre elas:

1. O método deve trabalhar com uma distribuição arbitrária de nós;
2. O algoritmo deve ser estável, ou produzir resultados estáveis apesar de que algumas distribuições nodais poderem causar certa instabilidade na aproximação local;
3. O método deve convergir. Esta é uma característica particularmente importante neste trabalho, pois ao se aumentar o número de nós no domínio o método deve garantir que a solução seja melhor aproximada. Quando se trata de algoritmos paralelizados, a intenção é justamente obter soluções mais precisas (com menor erro) ao se diminuir o espaçamento entre os nós no domínio;
4. Os subdomínios, onde será avaliada a forma fraca local, devem ser compactos. Em outras palavras, eles devem ser suficiente pequenos se comparados ao domínio completo. Isso garante uma maior esparsidade da matriz global, e uma melhor eficiência na obtenção dos resultados. Esta também é uma característica interessante na paralelização do método na GPU, pois cada *thread*, ou linha de execução do programa que irá processar a avaliação da forma fraca em um único subdomínio, não precisará percorrer todos os nós no domínio para ter uma aproximação local da função de forma;
5. O algoritmo deve ter a mesma ordem de complexidade daquela do método dos elementos finitos;
6. É desejável, também, que as funções de forma possuam a propriedade da função delta de Kronecker, para fácil imposição das condições essenciais de contorno.

Em [13] são definidas a consistência e a compatibilidade das funções de forma. A consistência é a capacidade da aproximação reproduzir funções polinomiais de mais baixa ordem. Um método com consistência k pode reproduzir funções polinomiais de ordem até k. Por outro lado, a compatibilidade de um método se refere à continuidade da aproximação produzida nas fronteiras entre subdomínios.

Tanto a compatibilidade quanto a consistência são características importantes das funções de forma, pois afetam diretamente a precisão das soluções produzidas pelo MLPG. Neste trabalho, utilizam-se duas técnicas de geração de funções de forma comumente utilizadas na literatura do MLPG. O método dos mínimos quadrados móveis e o método RPIMp que utiliza o método PIM (do inglês *point interpolation method*) com funções de base radial (RBF's) enriquecidas com polinômios de primeira ordem.

3.3.1 Método dos mínimos quadrados móveis (MLS)

O método dos mínimos quadrados móveis é considerado um dos melhores esquemas para se aproximar dados com precisão. Uma característica interessante deste método é o fato de que a aproximação não passa através dos nós dados, este assume um conjunto de valores nodais fictícios e passa uma função suave por ele, como ilustrado na figura 3.4.

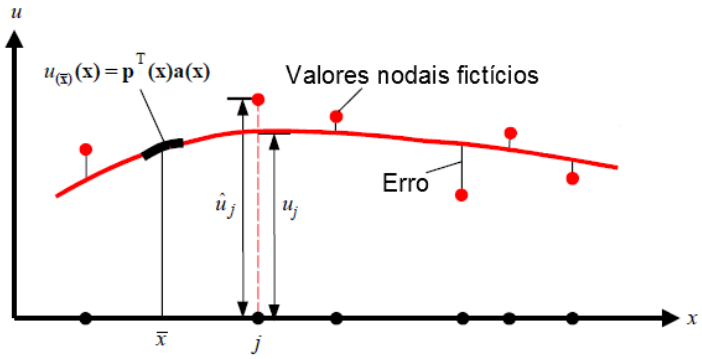


Figura 3.4: Interpolação com o MLS. Figura retirada de [14]

O MLS possui duas principais características: a primeira é que o método produz uma função aproximada contínua e suave em todo o domínio do problema; a segunda é que ele é capaz de produzir uma aproximação com qualquer ordem de consistência, bastando aumentar a ordem do polinômio base.

O MLS é utilizado no MLPG para gerar as funções de forma ϕ_i, considerando a aproximação de *u(x)* no ponto **x** como sendo $u^h(x)$. O MLS constrói essa função da seguinte maneira:

$$u^h(x) = \sum_j^m p_j(\mathbf{x}) a_j(\mathbf{x}) = \mathbf{p}^T(\mathbf{x})\mathbf{a}(\mathbf{x}) \tag{3.29}$$

Onde $a(x)$ é o vetor de coeficientes dado por:

$$a^T(x) = [a_0(x), a_1(x), ..., a_m(x)] \tag{3.30}$$

O termo $p(x)$ corresponde ao vetor de monomios de baixa ordem. Considerando o caso unidimensional tem-se:

$$p^T(x) = [p_0(x), p_1(x), p_2(x), ..., p_m(x)] = [1, x, x^2, x^3, ..., x^m] \tag{3.31}$$

No caso bidimensional,

$$p^T(x) = p^T(x, y) = [1, x, y, xy, x^2, y^2, ...] \tag{3.32}$$

E assim por diante, seguindo o triângulo de pascal, como ilustrado na figura abaixo:

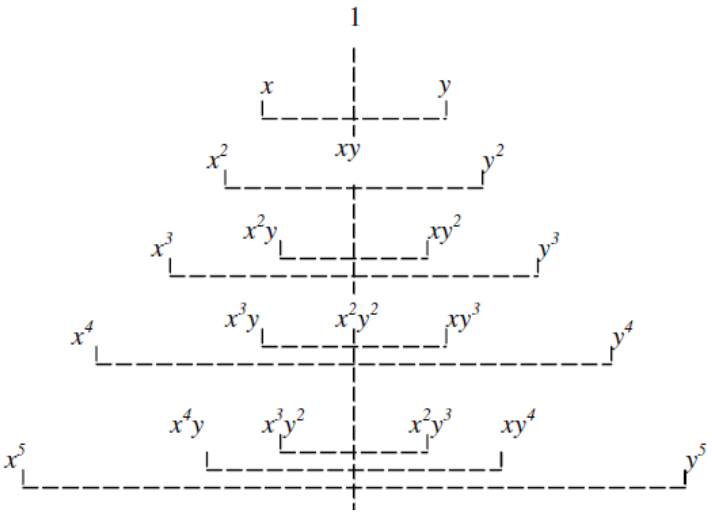


Figura 3.5: Triangulo de Pascal para a construção de p(x) em 2D. Figura retirada de [13]

O vetor de coeficientes $a(x)$ é determinado utilizando um conjunto de nós no domínio de suporte do ponto **x**. Os nós x_I no domínio de suporte de **x** são então utilizados da seguinte forma:

$$u^h(x, x_I) = p^T(x_I)a(x), I = 1, 2, 3, .., n \tag{3.33}$$

A função $a(x)$ aqui é uma função arbitrária em x. Um funcional peso residual é então construído utilizando os valores aproximados de u(x) e também os parâmetros nodais $u_I = u(x_I)$:

$$J = \sum_I^n W(x - x_I)[u^h(x, x_I) - u(x_I)]^2 \tag{3.34}$$

$$J = \sum_I^n W(x - x_I)[p^T(x_I)a(x) - u(x_I)]^2 \tag{3.35}$$

Onde $W(x - x_I)$ é uma função peso. Essa função tem o papel de garantir que diferentes nós têm diferentes pesos, e geralmente se prefere que nós mais distantes do ponto de aproximação tenham pesos menores que nós mais próximos. Outro papel também muito importante é o de garantir que os nós entram e saem do domínio de suporte de forma suave quando se move o ponto de aproximação *x*. Essa última garante a condição de compatibilidade **??**.

A aproximação pelo MLS é tal que para um dado ponto x, pretende-se encontrar os coeficientes **a(x)** que minimizem o funcional *J*.

$$\frac{\partial J}{\partial \mathbf{a}} = 0 \tag{3.36}$$

Isto resulta no seguinte sistema de equações lineares.

$$\mathbf{A(x)a(x)} = \mathbf{B(x)}U_s \tag{3.37}$$

Onde **A(x)** é a matrix momento dada por:

$$A(x) = \sum_{I}^{n} W_I(x) p(x_I) p^T(X_I) \tag{3.38}$$

Onde

$$W_I(x) = W_I(x - x_I) \tag{3.39}$$

A matriz **B(x)** é dada por

$$B_I = W_I(x) p(x_I) \tag{3.40}$$

E U_s são os parâmetros nodais de cada nó presente no domínio de suporte:

$$U_s = [u_1, u_2, u_3, ..., u_n]^T \tag{3.41}$$

Solucionando a equação 3.37, para **a(x)** tem-se:

$$a(x) = A^{-1}(x) B(x) U_s \tag{3.42}$$

Substituindo o resultado acima na equação 3.29 obtém-se:

$$u^h(x) = \sum_{I}^{n} \sum_{j}^{m} p_j(s) A^{-1}(x) B(x)_{jI} u_I \tag{3.43}$$

Alternativamente,

$$u^h(x) = \sum_{I}^{n} \phi_I(x) u_I \tag{3.44}$$

Onde,

$$\phi_I(x) = \sum_{j}^{m} p_j(x) (A^{-1}(x) B(x)_{jI}) = \mathbf{p}^T \mathbf{A}^{-1} \mathbf{B}_I \tag{3.45}$$

O valor *m* é o número de termos no polinômio base, que geralmente é menor que *n*, o número de nós no domínio de suporte. Geralmente $n >> m$ para se evitar uma matriz de momento singular, ou seja para que A^{-1} exista. A derivada parcial $\phi_{I,k}$ pode ser obtida fazendo-se:

$$\phi_{I,k} = \sum_{j=1}^{m} [p_{I,k} (A^{-1} B)_{jI} + p_j (A^{-1} B_{,k} + A^{-}_{,k} 1 B)_{jI}] \tag{3.46}$$

A aproximação pelo MLS é bem definida quando se tem uma matriz A não singular. $\phi_I(x)$ é chamada de função de forma da aproximação pelo MLS, correspondente ao nó x_I. O fato de $\phi_I(x)$ se anular quando x não está no domínio de suporte do nó x_I preserva a característica

local do MLS. A suavidade da função de forma é determinada tanto pelo polinômio base, quanto pela função peso.

A escolha da função de peso é mais ou menos arbitrária, desde que a mesma seja positiva, e de suporte compacto. Em [7] as funções Gaussiana e Spline de quarta ordem são sugeridas. Neste trabalho utiliza-se a função Spline de quarta ordem definida por:

$$W_I(\mathbf{x}) = \begin{cases} 1 - 6(\frac{d_I}{r_I})^2 + 8(\frac{d_I}{r_I})^3 - 3(\frac{d_I}{r_I})^4 & 0 \leq d_I \leq r_I \\ 0 & d_I > r_I \end{cases} \tag{3.47}$$

Onde r_I é o raio de influência, e d_I é a distancia de x até x_I, ou seja, $d_I = |x - x_I|$. O raio r_I deve ser tal que contenha um número suficiente de nós no domínio de suporte que garante a não-singularidade da matriz de momentos $A(x)$, e deve ser pequeno o suficiente para preservar a característica local do método MLS. Ou seja, o domínio de suporte deve conter o mínimo necessário de nós para que seja possível realizar a aproximação.

3.3.2 Point Interpolation Method (PIM)

O PIM (ou do inglês point interpolation method) obtém uma aproximação fazendo com que a função interpolada passe pelos nós definidos dentro do domínio de suporte. A interpolação pelo PIM é dada da seguinte forma:

$$u^h(x, x_Q) = \sum_{i=1}^{n} B_i(x) a_i(x_Q) \tag{3.48}$$

Onde $u^h(x)$ é a aproximação da função *u*, obtida usando os valores de *u* nos nós do domínio de suporte de um ponto de interesse x_Q. $B_i(x)$ é a função base definida no espaço cartesiano $x^T = [x, y]$, n é o número de nós no domínio de suporte de x_Q, e $a_i(x)$ são os coeficientes da função de base $B(x)$ correspondente ao ponto x_Q.

O PIM foi desenvolvido utilizando-se dois tipos de funções base, o primeiro utiliza polinômios, e o segundo utiliza funções de base radial. Este último, tambem chamado de RPIM,será discutido na próxima seção, e uma variação do mesmo é utilizada neste trabalho.

3.3.3 Radial Point interpolation method (RPIM)

A intenção de se utilizar funções de base radial como função base no PIM é de contornar o problema da matriz de momento singular que acontece quando se utiliza de polinômios como função base [13]. É provado matematicamente que a matriz de momento do RPIM é sempre inversível para um número de pontos arbitrariamente distribuídos no domínio do problema.

O RPIM, portanto, utiliza funções de base radial no lugar das funções base $B(x)$, como na equação seguinte:

$$u^h(x, x_Q) = \sum_{i=1}^{n} R_i(x) a_i(x_Q) = R^T(x) a(x_Q) \tag{3.49}$$

Onde $a_i(x_Q)$ são os coeficientes para o ponto x_Q como definido anteriormente, $R_i(x)$ é a função de base radial com *r* sendo a distancia entre o ponto x ao nó x_i. Em duas dimensões *r* é definido por:

$$r = [(x - x_i)^2 + (y - y_i)^2]^{1/2} \tag{3.50}$$

O vetor $R(x)$ tem a seguinte forma:

$$R^T(x) = [R_1(x), R_2(x), R_3(x), ..., R_n(x)] \tag{3.51}$$

Pode-se citar diversas funções de base radial, com diferentes resultados na interpolação das funções de forma no MLPG. Em [13] o autor destaca quatro tipos de funções RBF's como aparece na tabela 3.2.

Tabela 3.2: Exemplos de funções de base radial.

Item	Nome	Expressão	Parametros
1	Multiquádrica (MQ)	$R_i(x,y) = (r_i^2 + C^2)^q = [(x-x_i)^2 + (y-y_i)^2 + C^2]^q$	C, q
2	Gaussiana (EXP)	$R_i(x,y) = \exp(-cr_i^2) = \exp(-c[(x-x_i)^2 + (y-y_i)^2])$	c
3	*Thin plate spline*	$R_i(x,y) = r_i^\eta = [(x-x_i)^2 + (y-y_i)^2]^\eta$	η
4	RBF logaritma	$R_i(r_i) = r_i^\eta \log r_i$	η

De maneira geral os parâmetros das funções RBF's são determinantes para uma boa aproximação da função utilizando o método RPIM. Uma otimização nesses parâmetros pode ajustar melhor o método para diferentes tipos de problemas.

Neste trabalho utiliza-se outro tipo de função RBF bem utilizada na literatura, que forma a categoria das RBF's de suporte compacto:

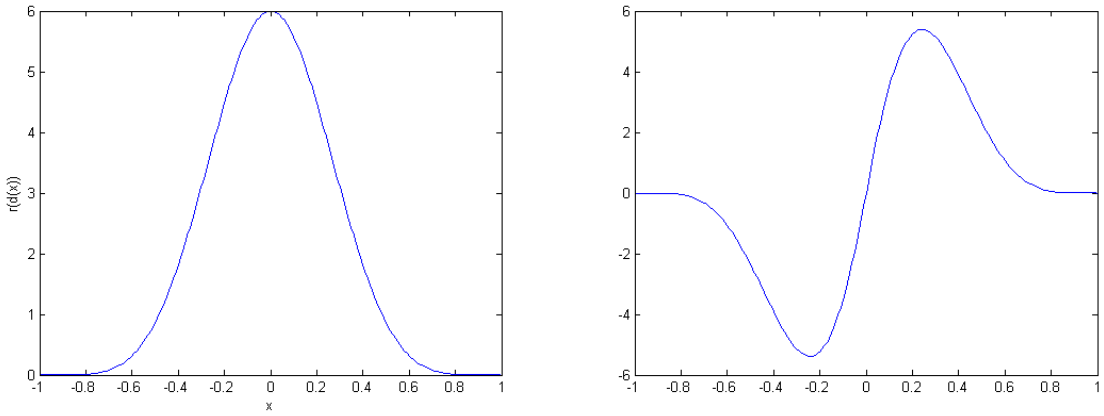
$$R(d) = \begin{cases} (1-d)^n p(d) & 0 \leq d \leq 1 \\ 0 & d > 1 \end{cases} \tag{3.52}$$

Onde $p(d)$ é um polinômio em função da distância do ponto ao nó central. Um exemplo é a função:

$$R(d) = \begin{cases} (1-d)^6(6 + 36d + 82d^2 + 72d^3 + 30d^4 + 5d^5) & 0 \leq d \leq 1 \\ 0 & d > 1 \end{cases} \tag{3.53}$$

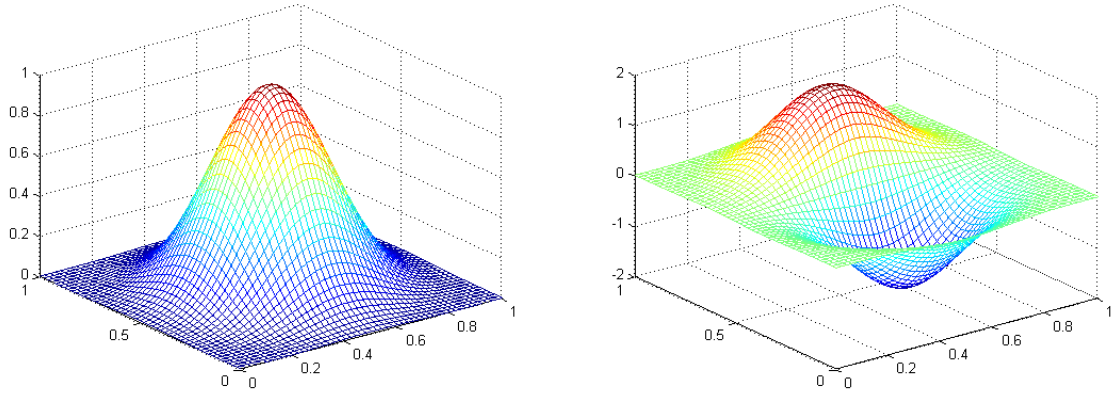
$$\frac{\partial R(d)}{\partial d} = 55d^{10} - 297d^8 + 693d^6 - 1155d^4 + 792d^3 - 88d, 0 \leq d \leq 1 \tag{3.54}$$

A equação acima representa a função de base radial conhecida como Wendland6 [9]. O seu formato é ilustrado nos gráficos da figura 3.6.



(a) (b)

Figura 3.6: Gráfico da função de base radial Wendland6 e sua derivada em 1-D.



(a) (b)

Figura 3.7: Gráfico da função de base radial Wendland6 e sua derivada em 2-D.

Uma vez escolhida a função RBF, a função $u(x)$ é aproximada pela equação 3.55 abaixo:

$$u^h(x, x_Q) = \sum_{i=1}^{n} R_i(x) a_i(x_Q) = R^T(x) a(x_Q) \tag{3.55}$$

Onde $a(x_Q)$ é o vetor de coeficientes desconhecidos e R_i é a função de base radial. Para problemas em duas dimensões, o parâmetro r_k pode ser determinado por:

$$r_k = [(x_k - x_i)^2 + (y_k - y_i)^2]^{1/2} \tag{3.56}$$

O vetor de coeficientes a_i 3.55 é determinado fazendo-se com que a aproximação passe por todos os n nós do domínio de suporte. Portanto, a aproximação para o k-ésimo ponto é dada por:

$$u_k = u(x_k, y_k) = \sum_{i=1}^{n} a_i R_i(x_k, y_k) k = 1, 2, 3, \cdots, n \tag{3.57}$$

Ou em formato matricial, tem-se:

$$U_s = R_Q \mathbf{a} \tag{3.58}$$

Onde R_Q é a matriz de momento e U_s é o vetor que determina o valor de campo em cada um dos *n* nós dentro do domínio de suporte.

$$R_Q = \begin{pmatrix} R_1(r_1) & R_2(r_1) & \cdots & R_n(r_1) \\ R_1(r_2) & R_2(r_2) & \cdots & R_n(r_2) \\ \vdots & \vdots & \ddots & \vdots \\ R_1(r_n) & R_2(r_n) & \cdots & R_n(r_n) \end{pmatrix} \tag{3.59}$$

Portanto, a solução para **a** existe e é única somente se a inversa de R_Q existe:

$$\mathbf{a} = R_Q^{-1} U_s \tag{3.60}$$

Substituindo-se a na equação 3.54 obtém-se:

$$u^h(x) = R^T(x)R_Q^{-1}U_s = \Phi(x)U_s \tag{3.61}$$

Onde $\Phi(\mathbf{x})$ é a matriz das funções de forma:

$$\Phi(x) = [R_1(x), R_2(X), \cdots, R_k(x), \cdots, R_n(x)]R_Q^{-1} = [\phi_1(x), \phi_2(x), \cdots, \phi_n(x)] \tag{3.62}$$

$$\phi_k = \sum_{i=1}^{n} R_i(x)S_{ik}^a \tag{3.63}$$

$$S_{ik}^a = R_{Q(ik)}^{-1} \tag{3.64}$$

As respectivas derivadas das funções de forma podem também serem deduzidas das expressões anteriores, como abaixo:

$$\frac{\partial \phi_k}{\partial x} = \sum_{i=1}^{n} \frac{\partial R_i}{\partial x} S_{ik}^a \tag{3.65}$$

$$\frac{\partial \phi_k}{\partial y} = \sum_{i=1}^{n} \frac{\partial R_i}{\partial y} S_{ik}^a \tag{3.66}$$

A matriz R_Q é sempre inversível desde que não se utilize certos parâmetros das RBF's que já são bem conhecidos. Esse fato é uma vantagem grande sobre o PIM utilizando-se somente polinômios. Porém, para certas distribuições de nós no domínio de suporte, pode se tornar a matriz R_Q^{-1} extremamente mal condicionada, o que de forma óbvia irá influenciar no erro da interpolação e, consequentemente, na solução do problema com MLPG. Entretanto, uma

vez escolhida uma boa distribuição de nós no domínio de suporte do nó de interess,e o RPIM já está ajustado ao problema, e sua influência no condicionamento da matriz global se torna invariável.

Pela definição de consistência da aproximação dada no capítulo 2, o RPIM não é consistente pois não reproduz polinômios lineares de maneira exata. Além disso, as funções de forma não satisfazem a propriedade da partição da unidade:

$$\sum_{i=1}^{n} \phi_i(x) = 1 \tag{3.67}$$

Porém adicionando-se polinômios de primeira ordem à base da função é possível obter um RPIM cujas funções de forma possuem consistência C^1. Este método é desenvolvido na próxima seção.

3.3.4 RPIM com polinômio (RPIMp)

Como dito na seção anterior o RPIM somente com funções de base radial não é consistente e tem dificuldades para aproximar campos lineares. A proposta de se adicionar polinômios à base da função é justamente ganhar a consistência do PIM com polinômios.

O RPIM com polinômios ou RPIMp aproxima o campo da seguinte forma:

$$u^h(x) = \sum_{i=1}^{n} R_i(x)a_i + \sum_{j=1}^{m} p_i(x)b_j = \mathbf{R}^T(\mathbf{x})\mathbf{a} + \mathbf{p}^T(\mathbf{x})\mathbf{b} \tag{3.68}$$

Onde $a_i(\mathbf{x})$ são os coeficientes da RBF $R_i(\mathbf{x})$, e b_j são os coeficientes da base polinomial $p_j(\mathbf{x})$. O número n é determinado pelo número de nós no domínio de suporte, porém o número m ou o número de monômios do polinômio base é escolhido de acordo com a ordem consistência a ser alcançada. Se for uma consistência C^1 um polinômio de primeira ordem é suficiente ($m = 3$ para caso 2-D). Geralmente se escolhe $m << n$ para melhor estabilidade das interpolações [13].

O vetor **a** na equação anterior é descrito como:

$$\mathbf{a}^T = [a_1, a_2, a_3, \cdots, a_n] \tag{3.69}$$

Enquanto o vetor de coeficientes **b** é determinado como:

$$\mathbf{b}^T = [b_1, b_2, b_3, \cdots, b_m] \tag{3.70}$$

O vetor de funções de base radial **R** é dado por:

$$\mathbf{R}^T(\mathbf{x}) = [R_1(\mathbf{x}), R_2(\mathbf{x}), R_3(\mathbf{x}), \cdots, R_n(\mathbf{x})] \tag{3.71}$$

E finalmente, o polinômio **p** é escrito como:

$$\mathbf{p}^T = [p_1(\mathbf{x}), p_2(\mathbf{x}), p_3(\mathbf{x}), \cdots, p_m(\mathbf{x})] \tag{3.72}$$

Os coeficientes $\mathbf{a}_i$ e $\mathbf{b}_j$ sao determinados forçando-se a interpolação passar por cada um dos *n* nós no domínio de suporte. A interpolação no ponto *k* é dada então por:

$$u_k(x_k, y_k) = \sum_{i=1}^{n} a_i R_i(x_k, y_k) + \sum_{j=1}^{m} b_j p_j(x_k, y_k), k = 1, 2, 3, \cdots, n \tag{3.73}$$

ou no formato matricial

$$\mathbf{U}_s = \mathbf{R}_Q \mathbf{a} + \mathbf{P}_m \mathbf{b} \tag{3.74}$$

Onde $\mathbf{U}_s$ é o vetor que armazena os valores de campo de todos os *n* nós no domínio de suporte. Para garantir a unicidade da aproximação a parcela polinomial da equação deve obedecer uma condição extra [13], para isso a seguinte restrição é imposta:

$$\sum_{i=1}^{n} p_j(x_i, y_i) a_i = 0, j = 1, 2, 3, \cdots, m \tag{3.75}$$

ou na forma matricial

$$\mathbf{P}_m^T \mathbf{a} = \mathbf{0} \tag{3.76}$$

O sistema de equações acima forma então um conjunto de equações homogêneas, que combinadas com a equação (3.74) resulta em:

$$\begin{pmatrix} \mathbf{R}_Q & \mathbf{P}_m \\ \mathbf{P}_m^T & \mathbf{0} \end{pmatrix} \cdot \begin{bmatrix} \mathbf{a} \\ \mathbf{b} \end{bmatrix} = \begin{bmatrix} \mathbf{U}_s \\ \mathbf{0} \end{bmatrix} \tag{3.77}$$

ou

$$\mathbf{G} \cdot \begin{bmatrix} \mathbf{a} \\ \mathbf{b} \end{bmatrix} = \begin{bmatrix} \mathbf{U}_s \\ \mathbf{0} \end{bmatrix} \tag{3.78}$$

Onde $\mathbf{G}$ é matriz de momento para o RPIMp. Logo uma única solução para $\mathbf{a}$ e $\mathbf{b}$ existe se a inversa de $\mathbf{G}$ existe:

$$\begin{bmatrix} \mathbf{a} \\ \mathbf{b} \end{bmatrix} = \mathbf{G}^{-1} \begin{bmatrix} \mathbf{U}_s \\ \mathbf{0} \end{bmatrix} \tag{3.79}$$

Para se obter uma solução para $\mathbf{a}$ e $\mathbf{b}$, parte-se portanto da equação (3.74).

$$\mathbf{a} = \mathbf{R}_Q^{-1}\mathbf{U}_s - \mathbf{R}_Q^{-1}\mathbf{P}_m\mathbf{b} \tag{3.80}$$

Substituindo a equação acima na equação (3.76) obtém-se:

$$\mathbf{P}_m^T(\mathbf{R}_Q^{-1}\mathbf{U}_s - \mathbf{R}_Q^{-1}\mathbf{P}_m\mathbf{b}) = 0 \tag{3.81}$$

$$\mathbf{P}_m^T\mathbf{R}_Q^{-1}\mathbf{U}_s = \mathbf{P}_m^T\mathbf{R}_Q^{-1}\mathbf{P}_m\mathbf{b} \tag{3.82}$$

$$\mathbf{b} = (\mathbf{P}_m^T\mathbf{R}_Q^{-1}\mathbf{P}_m)^{-1}\mathbf{P}_m^T\mathbf{R}_Q^{-1}\mathbf{U}_s \tag{3.83}$$

$$\mathbf{b} = S_b\mathbf{U}_s \tag{3.84}$$

Onde S_b é dado por:

$$S_b = (\mathbf{P}_m^T\mathbf{R}_Q^{-1}\mathbf{P}_m)^{-1}\mathbf{P}_m^T\mathbf{R}_Q^{-1} \tag{3.85}$$

Substituindo a equação (3.84) na equação (3.80) obtém-se a solução para **a**:

$$\mathbf{a} = \mathbf{S}_a\mathbf{U}_s \tag{3.86}$$

Onde

$$\mathbf{S}_a = \mathbf{R}_Q^{-1}[\mathbf{I} - \mathbf{P}_m\mathbf{S}_b] = \mathbf{R}_Q^{-1} - \mathbf{R}_Q^{-1}\mathbf{P}_m\mathbf{S}_b \tag{3.87}$$

Finalmente substituindo-se **a** e **b** na equação (3.68), obtém-se a solução para $u(x)$:

$$u^h(x) = [\mathbf{R}^T(\mathbf{x})\mathbf{S}_a + \mathbf{p}^T(\mathbf{x})\mathbf{S}_b]\mathbf{U}_s \tag{3.88}$$

Portanto, a matriz de funções de forma pode ser obtida por:

$$\Phi(\mathbf{x}) = [\mathbf{R}^T(\mathbf{x})\mathbf{S}_a + \mathbf{p}^T(\mathbf{x})\mathbf{S}_b] = [\phi_1, \phi_2, \phi_3, \cdots, \phi_n] \tag{3.89}$$

Onde ϕ_i é a função de forma para o nó *i*, dada por:

$$\phi_k(x) = \sum_{i=1}^{n} R_i(x)S^a_{ik} + \sum_{j=1}^{m} p_j(x)S^b_{jk} \tag{3.90}$$

Onde S^a_{ik} é o elemento (i,k) de S_a, e S^b_{jk} o elemento (j,k) de S_b, que por sua vez são matrizes constantes para um dado nó *k*. As respectivas derivadas $\frac{\partial \phi_k}{\partial x}$ e $\frac{\partial \phi_k}{\partial y}$ podem ser então escritas, para o caso 2-D, como:

$$\frac{\partial \phi_k}{\partial x} = \sum_{i=1}^{n} \frac{\partial R_i}{\partial x}S^a_{ik} + \sum_{j=1}^{m} \frac{\partial p_j}{\partial x}S^b_{jk} \tag{3.91}$$

$$\frac{\partial \phi_k}{\partial y} = \sum_{i=1}^{n} \frac{\partial R_i}{\partial y}S^a_{ik} + \sum_{j=1}^{m} \frac{\partial p_j}{\partial y}S^b_{jk} \tag{3.92}$$

Para o RPIMp, como descrito nesta seção, deve-se obter a inversa de $[\mathbf{P}^T_m\mathbf{R}_Q\mathbf{P}_m]$ que , diferentemente do RPIM com funções RBF puro, não é sempre inversível. Porém, para situações práticas, fazendo-se $n >> m$ é sempre possível obter $[\mathbf{P}^T_m\mathbf{R}_Q\mathbf{P}_m]^{-1}$ [13].

3.4 Imposição das condições de contorno de Dirichlet

No MLPG com MLS a imposição das condições de contorno essenciais ou de Dirichlet não pode ser efetuada diretamente como no MLPG+RPIMp ou RPIM. Portanto, técnicas adicionais são necessárias para impor as condições de contorno. Isso se deve ao fato das funções de forma no MLS não obedecerem a propriedade da função de delta de Kronecker:

$$\phi_j(x_k) \neq \delta_{jk} \tag{3.93}$$

Considerando as formas fracas desenvolvidas nas seções 3.1 e 3.2, reescritas a seguir por conveniência:

$$\int_{\Omega_s} b\frac{du}{dx}\frac{dv}{dx}dx + \int_{\Omega_s} cvudx - \int_{\Omega_s} fvdx - [vq]_{\Gamma_{su}} - [v\tilde{q}]_{\Gamma_{sq}} + \alpha_u[(u-\tilde{u})v]_{\Gamma_{su}} = 0 \tag{3.94}$$

$$\int_{\Gamma_s u} \hat{n}\cdot\nabla u d\Gamma + \int_{\Gamma_s q} \hat{n}\cdot\nabla u d\Gamma + \int_{\Gamma_s u} \hat{n}\cdot\nabla u d\Gamma + \int_{\Gamma_s u} \alpha(u-\tilde{u})d\Gamma - \int_{\Omega_s} pd\Omega = 0 \tag{3.95}$$

o termo que contém α é justamente o termo adicionado à forma fraca local do MLPG com o intuito de impor as condições de Dirichlet.

$$\int_{\Gamma_s u} \alpha(u-\tilde{u})d\Gamma \tag{3.96}$$

Uma dificuldade no método MLPG+MLS é justamente encontrar um valor de α que funcione para vários problemas físicos diferentes.

Capítulo 4

Arquitetura unificada ou CUDA

Os processadores gráficos evoluíram a partir de um hardware com pipeline de estágios fixos a um processador altamente paralelo com pipeline programável. As primeiras GPUs continham um pipeline que consistia de dois estágios programáveis, o primeiro era o processador de vértices que executava os programas de *shaders* de vértices e o segundo era o processador de pixel que, por sua vez, executa os programas de *shaders* de pixels. Um *shader* é um programa que calcula e produz a quantidade certa de luz e cor para uma determinada imagem [15], enquanto pixels e vértices são primitivas muito comuns em processamento gráfico. Em novembro de 2006 a NVIDIA introduziu a GPU GeForce 8800, que unificou os processadores de vértices e de pixel, abrindo caminho também para a programação de aplicações paralelas com a arquitetura CUDA ou do inglês *Computer Unified Architecture*, o que habilitou o desenvolvimento de ferramentas em linguagem C para a GPU.

Neste capítulo apresenta-se uma rápida história da evolução das GPU's, com um esquema simples de como as GPU's chegaram ao conceito atual. Em seguida, discute-se a arquitetura *Tesla*, que abriu as portas para a programação de propósito geral na GPU, e também se mostra muito útil para explicar alguns conceitos básicos.Apresenta-se então o mapeamento de aplicações na arquitetura da placa de vídeo, para depois introduzir o modelo de programação na arquitetura CUDA. Após analisados alguns aspectos gerais sobre as GPU's, a arquitetura da GPU GeForce GTX 680 é analisada brevemente. Esta é utilizada para executar a aplicação do método sem malha e, portanto, se configura como o ambiente de teste e análise do trabalho em questão. Finalmente, apresenta-se como o método sem malha MLPG se encaixa na arquitetura da GPU, e qual será a abordagem utilizada neste trabalho.

4.1 Evolução das GPU's

Para conseguir atingir o seu propósito e renderizar gráficos de forma eficiente e cada vez mais rápido, o pipeline da GPU era dividido entre dois estágios no próprio hardware, que consistiam dos processadores de vértices e dos processadores de pixel-fragmento. Os processadores de vértices operavam sobre primitivas tais como pontos, linhas e triângulos. Uma operação comum incluía a transformação de coordenadas para o espaço da tela, e carregar os parâmetros para o cálculo de iluminação e textura a serem utilizados no próximo estágio. O processador de pixel-fragmento operava na saída da rasterização, o que preenchia o interior das primitivas através dos parâmetros interpolados.

Os processadores de vértices e de pixel-fragmento evoluíram a taxas diferentes e para propósitos diferentes. No entanto esse desenvolvimento separado não se mostrou vantajoso logo depois. Com a introdução da arquitetura Tesla, procurou-se unificar essas operações em um único conjunto de processadores, ou seja, ambas as operações sendo realizadas pelo mesmo hardware porém em estágios diferentes do processamento. Isto permite o balanceamento da carga do processamento de vértices e de pixels, e, com a introdução da *DirectX* 10 (DX10), permitiu a utilização dos *shaders* geométricos (os shader's geométricos é um estágio opcional introduzido pela primeira vez com a DX10, este estágio situa-se entre os estágios de *shader* de vértice e de pixel, e sua função básica é de criar novas primitivas a partir de primitivas já existentes [11]). Com o pipeline unificado, o foco da equipe de projeto do hardware se volta para um único processador rápido e eficiente, permitindo inclusive o compartilhamento de unidades de hardware consideradas caras, tais como as unidades de textura. A generalização requerida foi tamanha que possibilitou uma aplicação completamente nova, a programação paralela de propósito geral na GPU, [8].

As razões principais que direcionaram o desenvolvimento das GPU's aos estágios atuais foram permitir o balanceamento do processamento de pixels e vértices, mantendo todas as unidades funcionais do *hardware* ocupadas, o que tornou o custo do desenvolvimento mais efetivo. A programação de propósito geral na GPU veio de graça, como um efeito colateral que abriu novas oportunidades e desafios aos programadores.

Na tabela 4.1 é apresentada essa evolução através de exemplos de GPU's e o ano em a mesma foi lançada, assim como alguns detalhes principais que caracterizam essas GPU's, até se chegar

no XBox 360, que foi o primeiro processador gráfico unificado. Na seção seguinte a arquitetura Tesla, a sucessora na tabela 4.1, é estudada em detalhe.

Processador	Características principais
GeForce 256 (1999)	Processador de ponto flutuante de 32-bits, transformação de vértice e iluminação, pipeline de função fixa para fragmento de pixel. Programável em OpenGL e Microsoft DX7 API.
GeForce 3 (2001)	Processador de vértices programável. Pipeline de fragmentos em ponto flutuante de 32-bits configurável e programável com DX8 e OpenGL.
Radeon 9700 (2002)	Processador de pixel-fragmento de ponto flutuante 24-bits programável com DX9 e OpenGL.
GeForce FX (2003)	Processador de pixel-fragmento de ponto flutuante de 32-bit programável.
XBox 360 (2005)	Primeiro processador gráfico unificado, permitindo que vértices e pixels sejam calculados no mesmo processador.

Tabela 4.1: Evolução das GPU's com exemplos.

4.2 Arquitetura Tesla

A arquitetura Tesla é baseada num processador escalável organizado no formato de um vetor ou conjunto de unidades de processamento menores. A figura 4.1 ilustra a GPU GeForce 8800 que possui 128 núcleos de processadores de streaming (do inglês *streaming processors* - SP) e 16 multiprocessadores de streaming (*streaming multiprocessors* - SM's), presentes em oito unidades de processamento ou, como é chamado, cluster de processadores e texturas (do inglês *texture processing cluster* - TPC). O processamento flui de cima para baixo, começando com a interface com o *host* através do barramento PCI *express*.

A arquitetura Tesla será objeto de estudo nesta seção pelo fato de servir como base para o desenvolvimento dos processadores gráficos sucessores, e porque marcou o início do desenvolvimento do modelo de programação CUDA. Além de ser possível explicar os conceitos básicos desses processadores, sem entrar em muitos detalhes complexos presentes na atuais arquiteturas.

O array de processadores de streaming (ou do inglês *streaming processor array* - SPA) realiza todos os cálculos da GPU programável, esta unidade é aquela que inclui todos os TPC's na figura 4.1. O SPA não inclui a unidade ROP (ou *raster operation processor*), que é um buffer de memória para operações de cor e profundidade diretamente na memória DRAM. E também não inclui todo o controle da memória DRAM no chip. O restante das unidades entregam trabalhos ou alimentam o SPA.

A unidade *input assembler* coleta todas as operações a serem realizadas sobre os vértices, e os respectivos trabalhos são distribuídos pela unidade seguinte, denominada *vertex work distribution*. Os TPC's executam os códigos dos programas de *shader* para vértices e para geometrias, além de armazenar os resultados desta operação em um *buffer* de memória no próprio *chip* que, por sua vez, passa os resultados para a unidade de *Viewport/clip/setup/raster/zcull*. Esta unidade toma esses resultados e rastreia em fragmentos de pixel. A unidade de distribuição de trabalhos de fragmentos de pixel, por sua vez, distribui todo o trabalho de processamento de fragmento de pixel entre os TPC's. Consequentemente, os fragmentos de pixel dessa vez calculados são enviados através das interconexões para processamento de profundidade e cor pelas unidades ROP. Por último, a unidade de trabalho de distribuição de computação (*Compute work distribution*), distribui tarefas gerais de computação aos TPC's, ou seja, esta é a unidade responsável pela distribuição das computações de propósito geral.

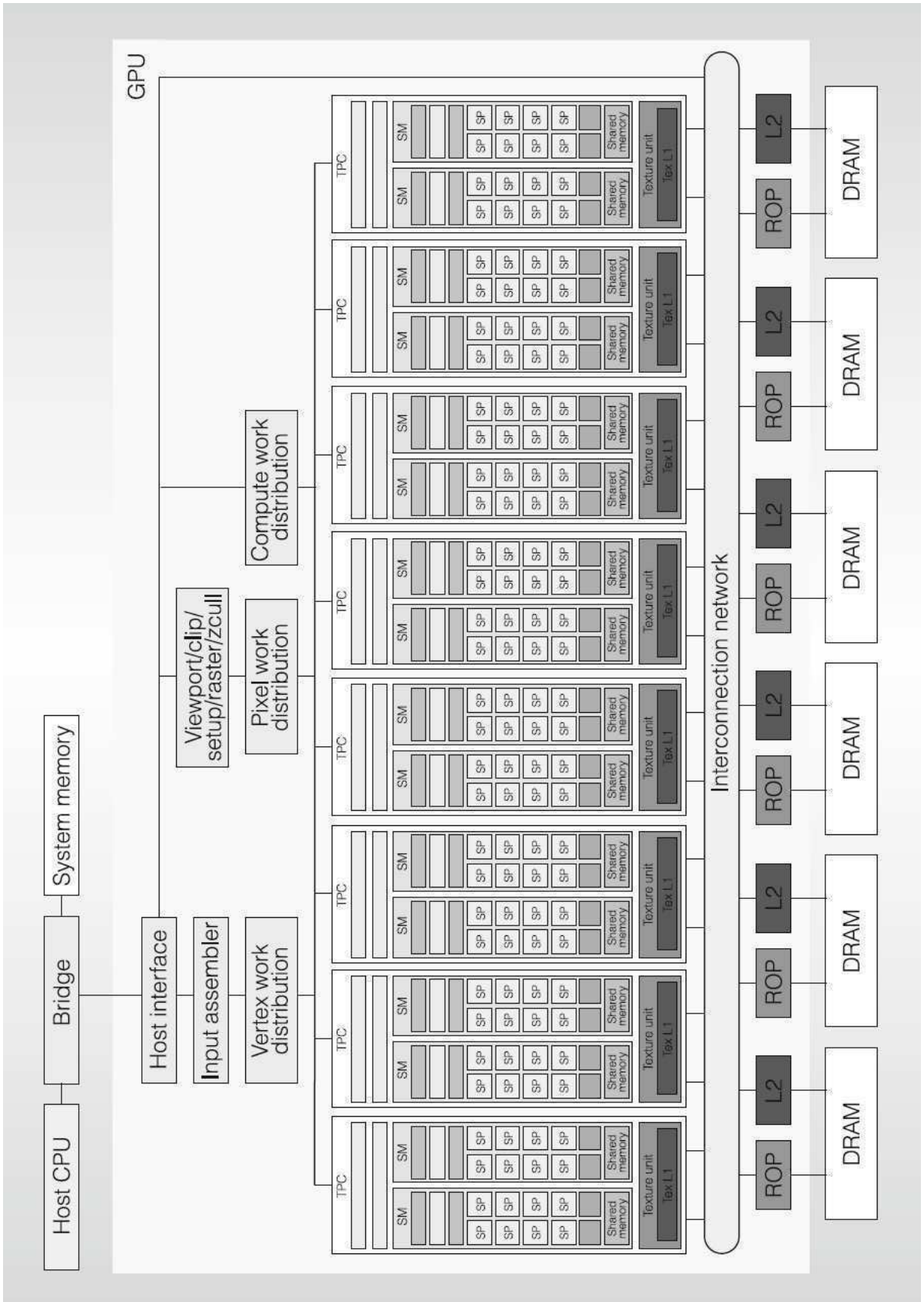


Figura 4.1: GPU arquitetura Tesla com pipeline unificado. Erik Lindholm [8].

Tanto a unidade *Compute work distribution* quanto *Vertex work distribution* distribuem as tarefas num esquema round-robin **??**, entretanto a unidade de *pixel work distribution* distribui as tarefas de pixels de acordo com a localização dos mesmos.

O SPA executa os programas na GPU além de gerenciar todas as *threads* e também fornecer algumas diretivas de controle para o programador. O número de TPC's é o que determina a performance das aplicações, as GPU's podem conter duas TPC's, oito ou até mais. Essa característica torna a arquitetura escalável, bastando somente aumentar o número de TPC's. Na figura 4.2 pode-se ver o TPC em detalhes.

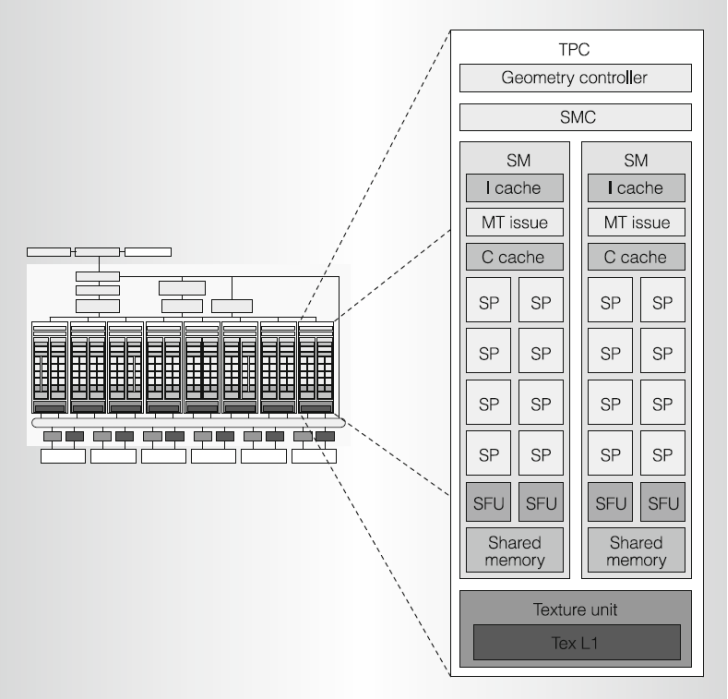


Figura 4.2: Texture/processor cluster ou TPC. Figura retirada de [8].

A primeira unidade que aparece no TPC, o controlador de geometria, gerencia entradas e saídas no chip, atributos de armazenamento de vértices e aloca conteúdo quando necessário. Operações típicas são transformação de coordenadas de posição e cor, e a geração de coordenadas de textura.

A unidade SM, do inglês *Streaming Multiprocessor*, é a unidade que executa tanto o código de processamento de vértices, geometrias e também de fragmento de pixels. Ela é responsável

também pela execução de programas de propósito geral paralelos. Esta unidade possui uma memória compartilhada que pode ser utilizada para armazenar dados utilizados por *threads* concorrentes.

Para executar *threads* em paralelo de maneira eficiente enquanto rodam vários programas diferentes, o SM é um *hardware multithreaded*, que consegue gerenciar até 768 *threads* concorrentes sem perdas com escalonamento.

Um desses multiprocessadores consiste de oito unidades funcionais (ilustrado na figura 4.3) controladas por uma única unidade de decode de instrução. A unidade de decodificação de instrução é capaz de entregar uma instrução a cada quatro ciclos de clock, e por isso a menor unidade de execução é o que a NVIDIA chama de *warp*, um conjunto de 32 *threads*.

Cada SM gerencia um conjunto de 24 *warps* com um total 768 *threads*. *Threads* que compõem um *warp* são do mesmo tipo e começam a execução a partir do mesmo endereço de instrução. Porém cada *thread* é livre para saltar e tomar caminhos de execução diferentes. Por isso a NVIDIA chama a arquitetura de *single-instruction, multiple thread*, ou SIMT, para diferenciar do conhecido *single-instruction, multiple data* ou SIMD.

Como um processador gráfico unificado, o SM executa diferentes *warps* de maneira concorrente (como por exemplo a execução de *warps* de vértices e *pixels* concorrentes). A cada ciclo o SM escolhe um dos 24 *warps* para executar uma instrução do tipo SIMT. Uma instrução de *warp* executa como dois conjuntos de 16 threads em quatro ciclos de processador.

Os núcleos SP (do inglês *streaming processor*) e as unidades SFU (*special function units*) executam instruções independentemente, entretanto instruções podem ser passadas entre ambos em ciclos alternados, de forma a mantê-los completamente ocupados. Os SP's são os núcleos que realizam o processamento das *thread*. Estes realizam operações fundamentais com ponto flutuantes. Como operações de multiplicação, adição, além de uma curiosa instrução de multiplicar e adicionar no mesmo ciclo (*multiply-add*). Ele também implementa uma gama de operações com inteiros, comparação e operações de conversão de tipo.

Já a unidade de função especial ou SFU é a unidade capaz de realizar computações de funções transcendentais e interpolação de atributos planos. Um processador gráfico tradicional

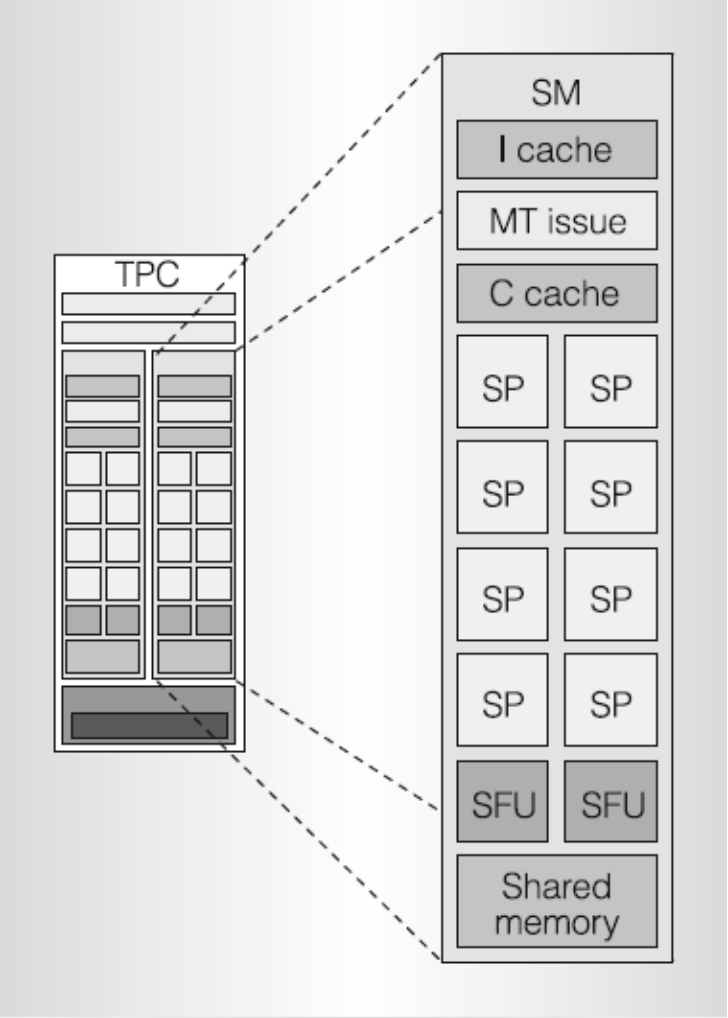


Figura 4.3: Streaming Multiprocessor em detalhe. Figura retirada de [8].

necessita de uma unidade para computar funções transcendentais dos algoritmos de *shader* de vértices e pixels. Pixels precisam de computar valores de atributos a uma dada localização (x,y), uma vez que se obtém os atributos a partir de primitivas ou dos vértices. Uma unidade funcional é capaz de gerar um número em ponto flutuante de 32-bits a cada ciclo de *clock*.

Para suportar computação e programação em linguagens C/C++, o SM da arquitetura *Tesla* implementa operações de *load/store* de memória juntamente ao carregamento de textura e saída de pixel do gráfico. Instruções de *load/store* de memória utilizam um endereçamento de *byte* inteiro, ou seja, um registrador e um offset, da mesma maneira que os compiladores tradicionais estão acostumados.

Existem três tipos de memória de escrita e leitura que uma instrução de *load* ou *store* pode acessar:

- Memória local por thread, privada e temporária (implementada em DRAM externa);
- Memória compartilhada de baixa latência de acesso para *threads* cooperativas na mesma SM (Memória no chip ou registradores), acessível somente por *threads* de mesmo bloco;
- Memória global para dados compartilhados entre *threads* de toda a aplicação(DRAM externa).

Para otimizar o acesso à memória e melhorar o uso da taxa de transferência, o acesso à memória pelo *load/store* global é coalescido, ou seja, todo acesso de uma *thread* individual é agrupado com o acesso de todas as *threads* de um mesmo bloco, para se ter um menor número de acessos à memória global. Por isso é interessante que *threads* do mesmo bloco acessem uma mesma região contígua de memória.

4.3 Mapeamento de aplicações para arquitetura da GPU

Como dito anteriormente a arquitetura Tesla permite que tanto aplicações gráficas quanto computações de propósito geral paralelas sejam executadas no mesmo processador. Essas aplicações têm algumas propriedades que as distinguem das aplicações na CPU, como listado abaixo:

- extensivo paralelismo de dados - milhares de computações em elementos de dados diferentes;
- paralelismo de tarefas modesto - grupos de threads executam o mesmo programa, enquanto diferentes grupos podem rodar diferentes programas;
- tolerância a latência - a performance de uma aplicação é definida somente pelo tempo do trabalho completo realizado;

- *streaming* de dados - requer alta taxa de largura de banda de memória e pouca reutilização de dados;
- comunicação entre threads e sincronização modesta - threads no processador gráfico não se comunicam, e aplicações paralelas devem requer sincronização e comunicação limitadas.

De forma a mapear um problema efetivamente em uma arquitetura CUDA, o programador deve decompor o problema em vários problemas pequenos que podem ser resolvidos paralelamente. Por exemplo, o problema pode ser partilhado em grupos maiores, que podem ser executados independentemente sem comunicação ou sincronização (chamado de *grid*), que por sua vez são divididos em elementos de execução ou *blocos* menores que são executados concorrentemente, esses blocos podem ser ainda divididos em *threads* que podem se comunicar através de memória compartilhada e ter alguma sincronização. A figura 4.4 mostra como um problema é decomposto em um *grid* com *2x3* blocos, que por sua vez são decompostos em blocos com *5x3 threads*.

A partir dessa decomposição do programa em três níveis (*grids*, *blocos* e *threads*) é possível mapeá-lo de maneira direta para a arquitetura Tesla. Supondo que cada SM execute um bloco do resultado, *threads* são executadas em *warps* dentro de cada SP.

O programador então divide o algoritmo em *grids*, que por sua vez, são divididos em unidades mais granulares chamadas de blocos, que são computados em paralelo e independentemente. O bloco, por sua vez, é computado em unidades ainda mais granulares denominadas *threads* paralelas. As *threads* concorrentes podem ter algum tipo de comunicação, através de memória compartilhada, além de uma modesta sincronização do trabalho.

4.4 Modelo de programação CUDA

Cuda é uma pequena extensão da linguagem de programação C e C++. O programador escreve parte do programa de maneira serial, a ser executado na CPU, e outra parte de maneira paralela, para tomar vantagem da arquitetura da GPU. O programa serial então chama uma série de *kernels*, que podem conter uma simples função ou até mesmo programas inteiros. Esses *kernels* são executados na GPU e subdividem-se em blocos de threads (*kernels* são

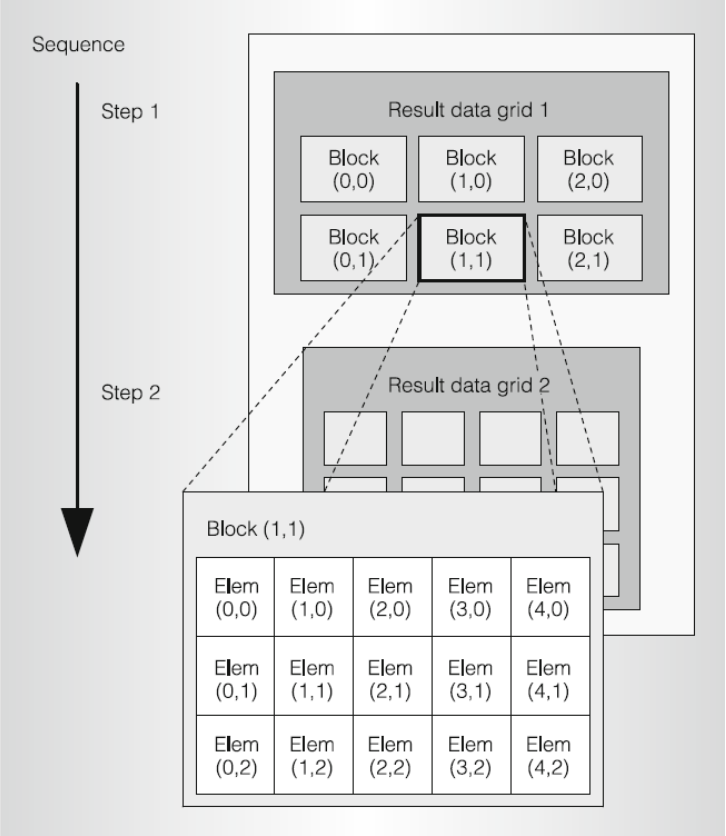


Figura 4.4: Decomposição do programa em um grid de bloco, e posteriormente em blocos de elementos computados em paralelo ou *threads*. Erik Lindholm [8].

equivalentes aos grids da seção anterior). O programador então organiza o código em uma hierarquia de *threads*, blocos de threads e grids, como descrito na seção anterior.

O compilador CUDA **nvcc** compila uma aplicação escrita em linguagem de programação C/C++ que possui um código integrado contendo tanto parte serial, que deve ser executada na CPU, quanto parte paralela, a ser executada na GPU.

A quantidade de SM's de uma GPU varia de acordo com o modelo, por exemplo uma placa de video do NVIDIA ION para computadores portáteis possui apenas duas SM's com oito núcleos cada (ou SP's). Já uma GPU GeForce GTX 680 possui 8 Multiprocessadores com um total de 192 núcleos em cada, totalizando 1536 núcleos. Apesar da diferença no processamento, a GPU é capaz de escalar a aplicação por todos esses núcleos independente do tipo da GPU, e de forma transparente para o programador, sem mesmo precisar de recompilar o código. Sendo

assim, a performance de um determinado programa pode ser medida através do número de núcleos presente em cada GPU, pelo menos para fins de comparação.

Na aplicação da figura 4.5, se tem um *grid* de oito blocos de *threads*. Numa GPU com dois SM's seriam distribuídos quatro blocos de *threads* para cada SM. Na GPU com 4 SM's seriam distribuídos dois blocos de *threads* para cada SM. Como a GPU é construída em cima de um *array* de SM's, e a aplicação é divida entre blocos de *threads*, a aplicação se encaixa em qualquer GPU com esta arquitetura de maneira direta, desde que a aplicação seja construída nesta hierarquia.

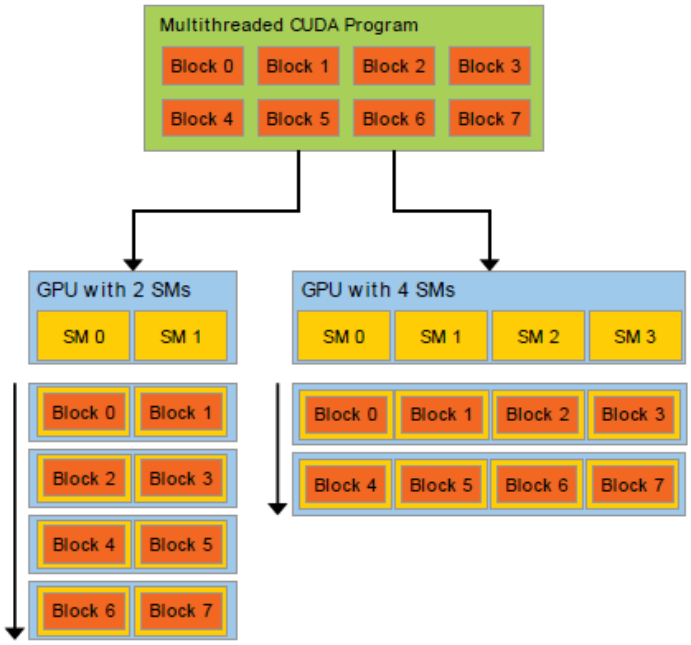


Figura 4.5: Decomposição do programa em blocos de threads e escalabilidade da GPU. GFL [5].

CUDA acrescenta algumas palavras reservadas, dentre as quais pode-se destacar: a declaração do *kernel* como **__global__** indicando que o código a seguir irá executar na GPU, além de representar o código principal que será chamado pela CPU; **__device__** que indica que determinada função ou procedimento deve ser compilado para a GPU; e **__shared__** para declaração de dados que são armazenados na memória compartilhada de cada SM, e portanto pode ser utilizado entre *threads* do mesmo bloco.

Um *kernel* em CUDA é simplesmente uma função em C, para o código sequencial de uma única *thread*. Esta função é compilada para GPU e chamada a partir da CPU, utilizando a seguinte sintaxe:

```
1 __global__ kernel_name(list_of_parameters)
2 {
3 //procedimento do kernel
4 }
5 ...
6 //chamada do kernel a partir da CPU
7 kernel_name<<<nBlocks,nThreads>>> (list_of_parameters);
```

Na chamada do *kernel* são passados o número de blocos (*nBlocks*) dentro do grid de execução, e o número de *threads* por bloco (*nThreads*), além da lista de parâmetros passadas para a GPU. Quando invocado pela CPU, o kernel é executado por cada *thread* de execução, ou seja, $nBlocks * nThreads$ vezes.

4.5 GeForce GTX 680: arquitetura e propriedades

A GPU escolhida para realizar os testes neste trabalho foi a GPU GeForce GTX 680. Esta GPU tem um arquitetura ligeiramente diferente da arquitetura Tesla estudada em seções anteriores, porém os conceitos e definições são iguais. Nesta seção apresentam-se as diferenças e conceitos específicos da GeForce GTX 680.

A GeForce 680 possui uma arquitetura chamada *Kepler*, e é fabricada utilizando-se o processo de 28nm. Sua arquitetura é composta de quatro *clusters* de processamento gráficos (*Graphic Processing Clusters* - GPC), oito Multiprocessadores de *Streaming* (denominados *next generation multiprocessors* - SMX), e quatro controladores de memória, como pode ser observado na figura 4.6.

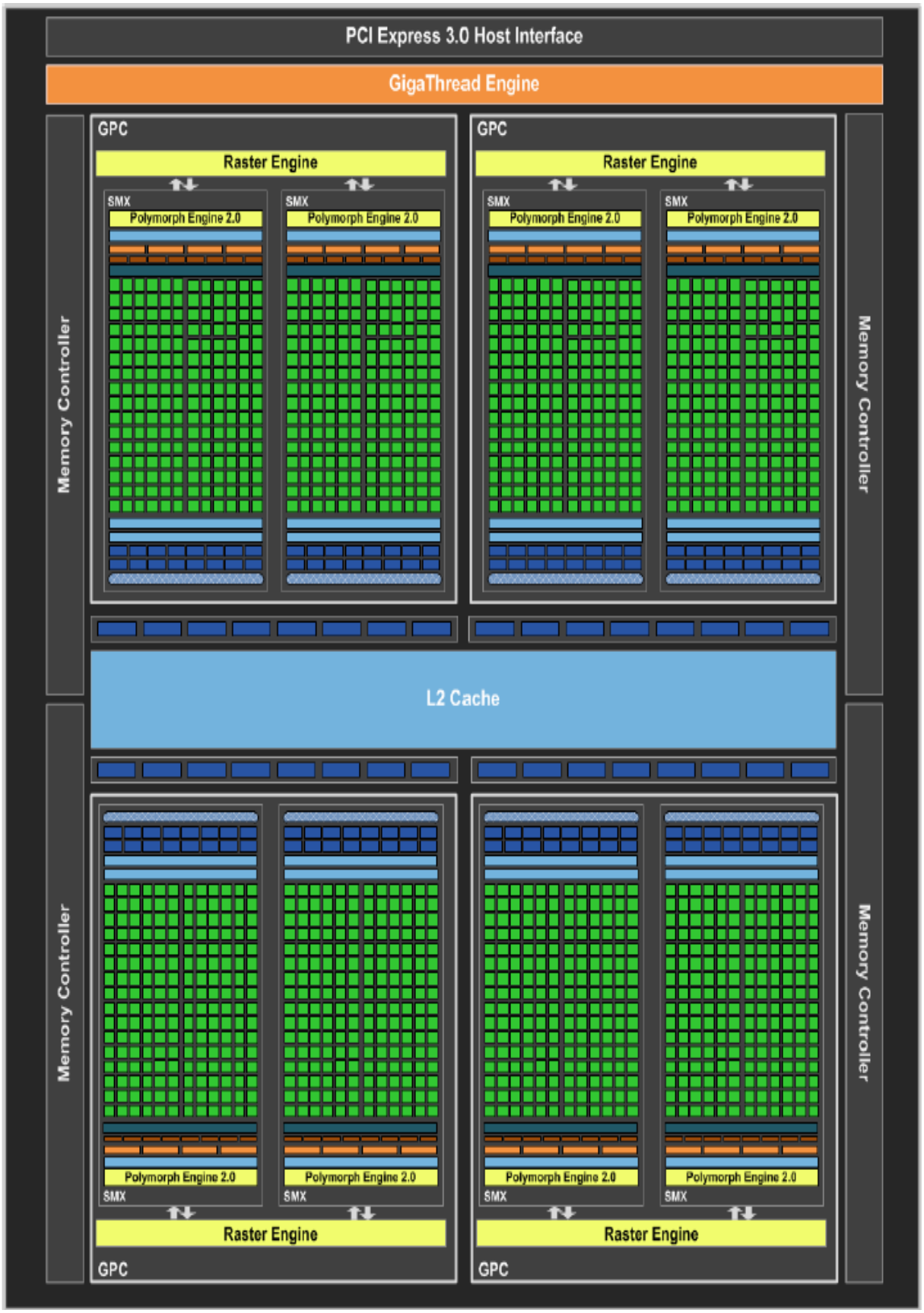


Figura 4.6: Arquitetura da GPU GeForce GTX 680. 680 [1].

Cada GPC tem uma unidade dedicada de rasterização além de dois SMX, totalizando oito SMX e 1536 núcleos CUDA (SP's). Conectado a cada controlador de memória existe um cache L2 de 128KB e oito unidades ROP. Portanto, com quatro controladores de memória, a GTX 680 possui um total de 512KB de cache L2 e 32 unidades ROP, para amostragem de cor.

Na tabela 4.2 comparam-se alguns parâmetros das GPU's, partindo-se da arquitetura Tesla, passando pela Fermi até chegar na Kepler em questão.

Tabela 4.2: Comparação entre as arquiteturas Tesla, Fermi e Kepler. 680 [1].

GPU	GT200 (Tesla)	GF110 (Fermi)	GK104 (Kepler)
Transistors	1.4 billion	3.0 billion	3.54 billion
CUDA Cores	240	512	1536
Graphics Core Clock	648MHz	772MHz	1006MHz
Shader Core Clock	1476MHz	1544MHz	n/a
GFLOPs	1063	1581	3090
Texture Units	80	64	128
Texel fill-rate	51.8 Gigatexels/sec	49.4 Gigatexels/sec	128.8 Gigatexels/sec
Memory Clock	2484 MHz	4008 MHz	6008MHz
Memory Bandwidth	159 GB/sec	192.4 GB/sec	192.26 GB/sec
Max # of Active Displays	2	2	4
TDP	183W	244W	195W

4.6 Método MLPG na GPU

O método MLPG é em essência paralelo. Uma vez obtida a forma fraca local, esta deve ser avaliada em cada subdomínio. Nesta seção apresenta-se um esquema geral de paralelização do algoritmo MLPG, independente do método de interpolação utilizado para aproximar as funções de forma. Logo depois, será abordada a questão da paralelização nos casos onde se utiliza o método dos mínimos quadrados móveis e o método RPIM com polinômio de primeiro grau separadamente, destacando-se as suas vantagens e desvantagens.

4.6.1 Esquema geral

O MLPG pode ser dividido em duas etapas principais, a primeira é a montagem do sistema de matrizes, através da avaliação da forma fraca local em cada subdomínio, e a segunda é a solução do sistema de equações e a computação dos resultados nos pontos de interesse.

Na figura 4.7 é apresentado um diagrama de fluxo que resume o método MLPG em diferentes etapas. A etapa destacada em azul no diagrama, representa a parte do algoritmo a ser executada em paralelo (a montagem do sistema de equações). A segunda parte é representada pelas duas etapas finais do algoritmo e será executada de maneira serial na CPU.

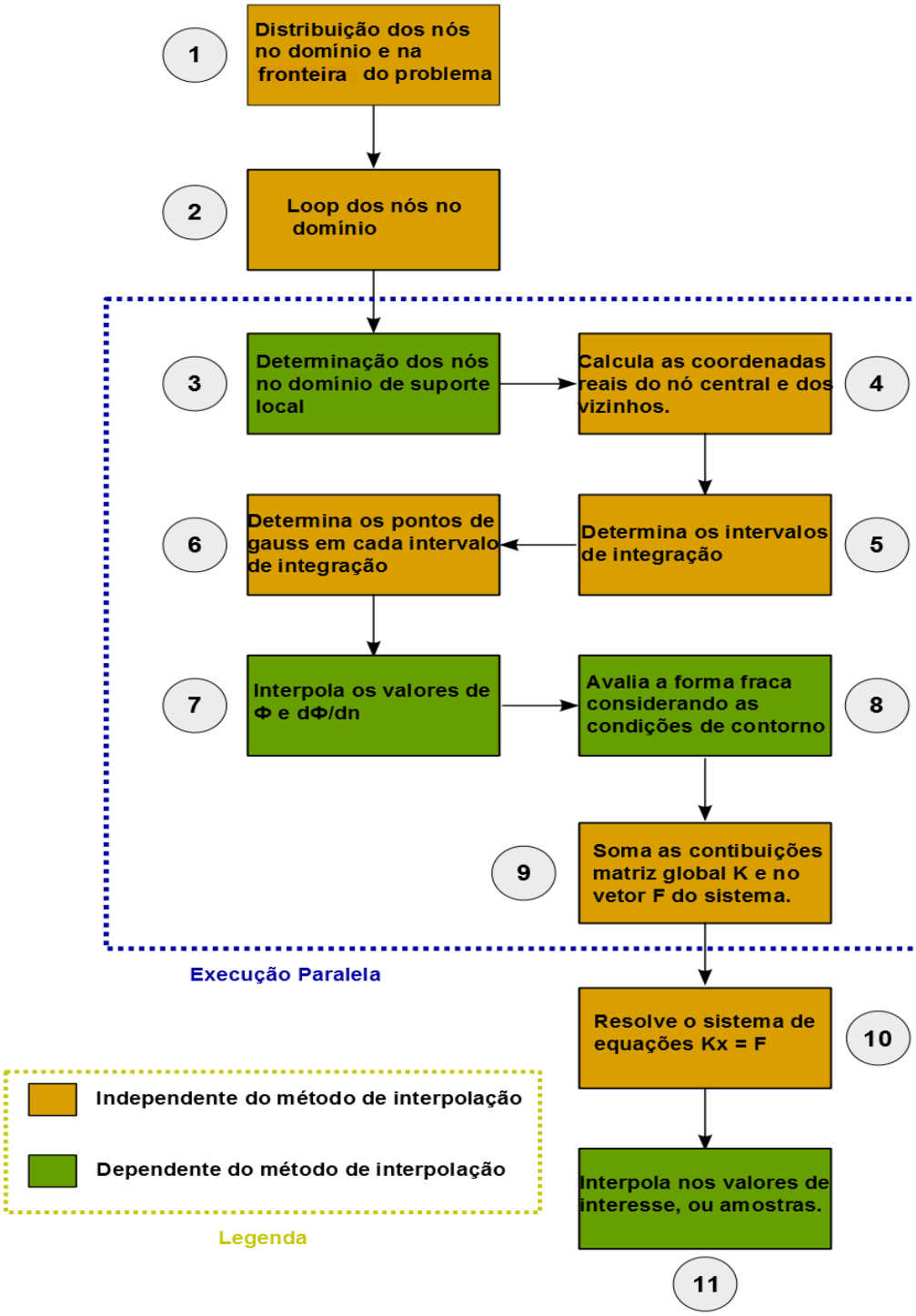


Figura 4.7: Esquema geral do MLPG com destaque para as partes que podem ser executadas em paralelo neste trabalho e para aquelas partes dependentes do método de aproximação das funções de forma.

A figura 4.7 também ressalta as partes do algoritmo que são dependentes do método de interpolação das funções de forma, ou seja, essas são as etapas que sofrerão alguma modificação dependendo do método escolhido.

A montagem do sistema de equações é a parte que mais consome processamento ou a mais cara de todo o processo, uma vez que, a forma fraca deve ser avaliada para cada nó do subdomínio do problema calculando os nós no domínio de suporte, interpolando os valores nos pontos de Gauss para finalmente realizar a integração numérica e somar a contribuição na matriz global. Essa é uma excelente candidata à ser executada em paralelo.

A próxima etapa, que é a solução do sistema global e interpolação dos respectivos valores nodais nos pontos de interesse (onde se quer saber o valor do campo). Apesar de existirem diversos métodos computacionais paralelos para se resolver este sistema, neste trabalho prefere-se resolver o sistema de equações global de maneira serial. Esta etapa não será considerada neste trabalho, concentrado-se apenas na montagem do sistema por ser a etapa mais cara do processo, e por proporcionar material de estudo para melhoria e adaptação do método MLPG, como será evidenciado no capítulo 5.

Nesta seção explica-se cada uma das etapas numeradas na figura 4.7, seguindo a ordem em que são executadas pelo algoritmo:

1. A primeira etapa é a distribuição dos nós no interior do domínio e na fronteira. O algoritmo pode também receber as informações da posição dos nós, juntamente com a descrição das fronteiras e as condições de contorno a partir de um arquivo externo. Esta parte é executada na CPU, os nós distribuídos no domínio são alocados dinamicamente e copiados para a memória da GPU. Em seguida
2. Em seguida, vem a etapa de inicialização que inclui: a alocação da memória na GPU para armazenamento do sistema de equações, e a inicialização de várias constantes e parâmetros do método. Após a inicialização, um *loop* entre os nós do problema deve ser realizado para a avaliação da forma fraca em cada subdomínio (caso na CPU). No caso da GPU, este *loop* não existe, pois ao chamar o *kernel* principal, um determinado número de *threads* é alocado de forma que cada uma delas irá computar uma avaliação de maneira concorrente (ou seja, tantas *threads* quantos nós houver no domínio do problema).

3. A parte três se resume em determinar o domínio de suporte ao nó local. Nesta etapa é feita uma varredura na vizinhança do nó para encontrar os nós que formam o domínio de suporte.

4. Na etapa quatro, as coordenadas reais (cartesianas) são calculadas. Isto é necessário dado que os nós são determinados e identificados utilizando uma numeração para facilitar o processo de busca.

5. A próxima etapa calcula a fronteira do subdomínio para realizar a integração numérica pelo método de Gauss-Legendre.

6. Após determinado os intervalos de integração, determinam-se também os pontos de integração.

7. Dando sequência, avaliam-se os valores de ϕ e $\frac{d\phi}{dn}$ nos pontos de integração, ou seja, calcula-se o valor das funções de forma e da sua derivada em relação à normal à fronteira do subdomínio, nos pontos de integração.

8. A forma fraca é avaliada, realizando-se todas as integrações numéricas. Nesta etapa devem ser consideradas as condições de contorno quando se utiliza o método da penalidade (caso do MLS).

9. Finalizando, as duas últimas etapas se resumem em resolver o sistema obtido, e calcular a função nos pontos de interesse. Esta etapa não é realizada na GPU, portanto as matrizes do sistema calculadas de maneira concorrente devem ser copiadas para a memória da CPU, antes da realização desta etapa.

4.6.2 Organização do código

A figura 4.8 apresenta um diagrama de classes UML para descrever a organização do código.

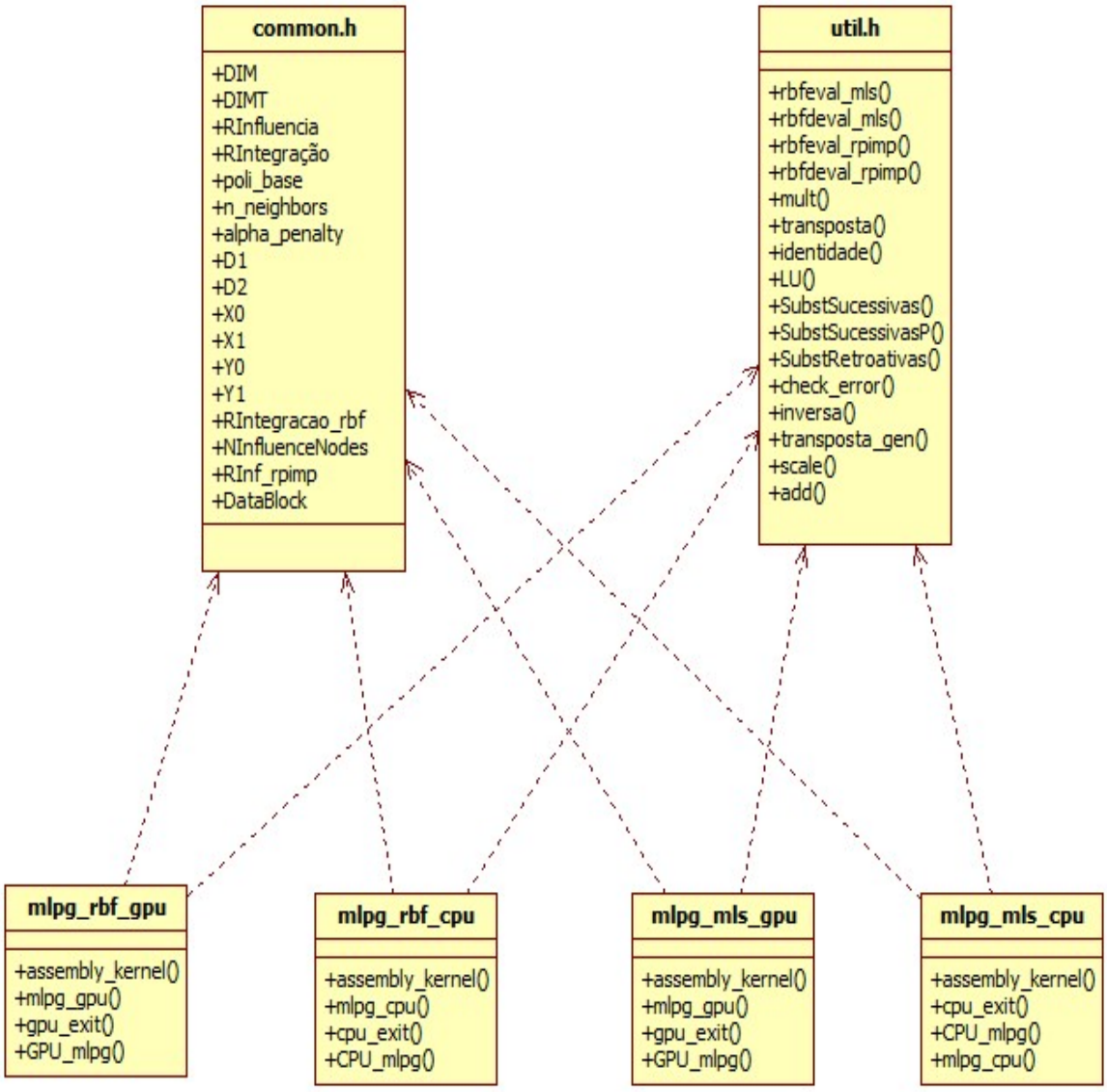


Figura 4.8: Diagrama geral da estrutura do algoritmo.

Existem dois arquivos fonte principais: a *common.h* que contém as constantes necessárias para se definir o problema, e a *util.h* que por sua vez contém os métodos utilizados durante o processo de montagem da matriz.

No arquivo *common.h* estão as definições das constantes que determinam o número de nós distribuídos uniformemente no domínio do problema (DIMT e DIM), os raios de influência e de integração (definidos separadamente para os métodos RPIMp e MLS), o tamanho do

vetor de base polinomial, as condições de contorno (D1 e D2), os limites do domínio do problema definidos como $[X0, X1]$ e $[Y0, Y1]$ e uma estrutura denominada *DataBlock*. Essa última é utilizada para armazenar a matriz K e o vetor F do sistema $Kx = F$, além das coordenadas dos nós no domínio do problema e mais um variável para armazenar o tempo gasto na montagem do sistema.

Já no arquivo *util.h* estão os procedimentos comuns necessários durante o processo de montagem do sistema, tais como: as funções de base radial (RBF) utilizadas no MLS e no RPIMp; funções de operações com matrizes (multiplicação, transposta, identidade, decomposição LU, substituições sucessivas e retroativas, função inversa, multiplicação por constante e soma de matrizes); e, além disso, um função importante para tratar os erros retornados pela GPU, a função *check_error*.

Essas funções genéricas têm a sua versão compilada para a GPU e para CPU, porém não é necessário replicar o código. Para isso, as funções são declaradas como no exemplo a seguir:

```
1 __device__ __host__ void inversa(double *A,double *Ainv)
2 {
3 //Função para obtenção da matriz inversa
4 }
```

As palavras reservadas $\_\_device\_\_$ e $\_\_host\_\_$ são utilizadas para indicar ao compilador que o procedimento deve ser compilado tanto para a GPU quanto para a CPU. Isso garante que o mesmo algoritmo é usado ao rodar na CPU ou na GPU, o que torna possível a comparação direta dos tempos de execução.

A partir desses dois arquivos fonte é possivel definir quatro outros arquivos que irão implementar o método sem malha, são eles: $mlpg\_rbfp\_cpu$, $mlpg\_rbfp\_gpu$, $mlpg\_mls\_cpu$ e $mlpg\_rbfp\_cpu$. Estes arquivos possuem ambos a mesma estrutura de funções como exemplificado na figura 4.9.

```
#include "common.h"
#include "util.h"

using namespace std;

namespace MLS_GPU{

    __global__ void assembly_kernel(double* data,double *K,double *F,int* IDs){ ... }

    void mlpg_gpu( DataBlock *d){ ... }

    // clean up memory allocated on the GPU
    void gpu_exit( DataBlock *d ){ ... }

    void GPU_mlpg(){ ... }

}
#endif
```

Figura 4.9: Estrutura dos arquivos *mlpg_mls_gpu.h* e *mlpg_rpimp_gpu.h*.

Cada um desses arquivos é responsável por implementar uma versão do método MLPG, o MLPG utilizando interpolação pelo MLS e o MLPG utilizando interpolação pelo RPIMp. Ambas as abordagens tem sua versão para a GPU e para a CPU. Na figura 4.10 segue a versão do codigo para a CPU.

```
#ifndef MLPG_CPU_H_INCLUDED
#define MLPG_CPU_H_INCLUDED

#include "common.h"
#include "util.h"

using namespace std;

namespace MLS_CPU{

    void assembly_kernel(double *K,double *F,int* IDs,int xx, int yy){ ... }

    void mlpg_cpu( DataBlock_cpu *d ){ ... }

    void cpu_exit(DataBlock_cpu *d){ ... }

    void CPU_mlpg(){ ... }

}
#endif
```

Figura 4.10: Estrutura dos arquivos *mlpg_mls_cpu.h* e *mlpg_rpimp_cpu.h*.

Cada um desses arquivos fonte possui quatro métodos: $assembly\_kernel$, $mlpg\_gpu$ ou $mlpg\_cpu$, cpu_exit ou gpu_exit, e CPU_mlpg ou GPU_mlpg. A função $assembly\_kernel$ executa a montagem do sistema de equações, esta é a função da GPU que é chamada pela CPU, por isso é declarada como $\_\_global\_\_$ nos arquivos fonte da GPU.

As outras funções desses arquivos são usadas somente para alocar memória a ser utlizada na GPU ou na CPU ($mlpg\_cpu$ ou $mlpg\_gpu$), para chamar a função $assembly\_kernel$ (CPU_mlpg ou GPU_mlpg) e para desalocar a memória alocada dinamicamente na GPU ou na CPU (CPU_exit ou GPU_exit).

Como pode ser notado na declaração da função $assembly\_kernel$ são passados os seguintes argumentos para a montagem do sistema de matrizes: a matriz K, o vetor F, um vetor de IDs para indexação dos nós do domínio além da estrutura $data$ com as coordenadas dos nós. Essa última só é passada para a GPU pois na CPU o acesso a esses dados é direto e via variável global. Vale a pena destacar o fato de que todos os dados são armazenados utilizando-se precisão dupla e que toda a memória alocada durante as operações matriciais da montagem do sistema de equações é feita estaticamente. Isso só é possível se se conhecer todos os tamanhos desses vetores e matrizes intermediárias em tempo de compilação, todos esses tamanhos já estão declarados no arquivo $common.h$ descrito anteriormente. Os únicos dados declarados dinamicamente são a matriz K, o vetor F, o vetor de IDs e a estrutura $DataBlock$, isso é feito por questões de performance.

Outra questão importante é o algoritmo para a busca dos nós vizinhos no domínio, ou a determinação do domínio de suporte (item 3 da figura 4.7). Nas seções seguintes cada uma das abordagens de busca é detalhada, a abordagem para o $MLPG+MLS$ e para o $MLPG+RPIMp$.

4.6.3 MLPG+MLS

Existem algumas diferenças importantes que influenciam na escolha do método de interpolação para o algoritmo MLPG utilizando o método de interpolação *Moving Least Squares* na GPU.

A primeira diferença importante aparece na etapa 3 do diagrama da figura 4.7. A determinação dos nós no domínio de suporte do MLPG+MLS é algo que deve ser feito com cautela, uma vez que cada nó no domínio tem um certo raio de influência, deve-se garantir que todos os

nós que influenciam o ponto de integração (onde se deseja avaliar a função de forma e sua derivada) estão incluídos nesta avaliação.

Na figura 4.11 é ilustrado um domínio onde os nós são uniformemente distribuídos, e se deseja avaliar a forma fraca num subdomínio qualquer (determinado pelo nó central). Os nós 1 e 2 fazem parte do domínio de suporte do nó central, pois o raio de influência dos mesmos englobam alguns pontos de integração. Por isso devem ser levados em conta na etapa 3 do MLS. Entretanto, o nó 3 não influencia nenhum ponto de integração deste subdomínio, por isso este não será incluído para avaliação na etapa 3.

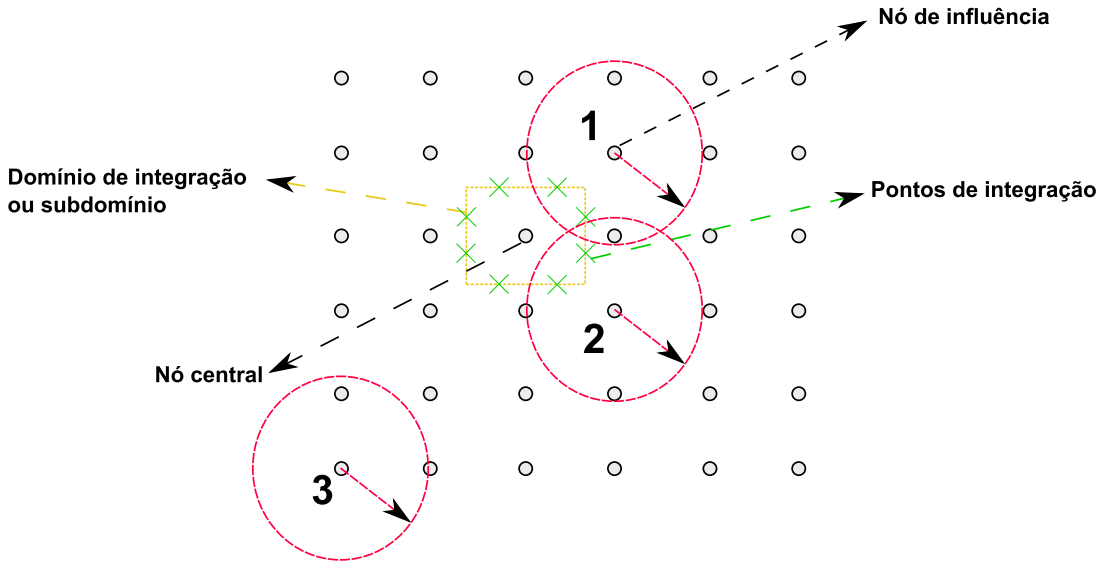


Figura 4.11: Ilustração dos domínios de influência dos nós, e determinação do domínio de suporte no MLS. Os nós 1 e 2 fazem parte do domínio de suporte do nó central, enquanto o nó 3 não.

No MLS, é necessário encontrar todos os possíveis candidatos a formar o domínio de suporte de cada nó. Como as *threads* na GPU são organizadas em *warps* de 32, o acesso à memória é então coalescido (como visto nas seções anteriores). Ao acessar a memória global é melhor que a memória acessada pelas *threads* esteja alinhada, pois essa região linear de memória (de 32, 64 ou 128 *bytes*) será copiada para um *cache* local do SM que está executando aquele *warp*. Consequentemente, ao realizar a busca pelos nós de influência, é interessante que todas as *threads* de um mesmo bloco acessem um porção contígua de memória [2].

Para isso o MLS necessita de um esquema diferente, que de certa forma só avalia os candidatos ao domínio de suporte localmente a cada subdomínio. Este esquema aparece na figura 4.12, onde se subdivide os nós no domínio em regiões, de forma que a varredura dos nós de influência se dá somente nas regiões vizinhas de cada nó central (ou subdomínio), evitando-se, portanto, o problema do acesso não-coalescido da memória global. A região em azul é a região onde a busca por nós no domínio de suporte do nó central se restringe.

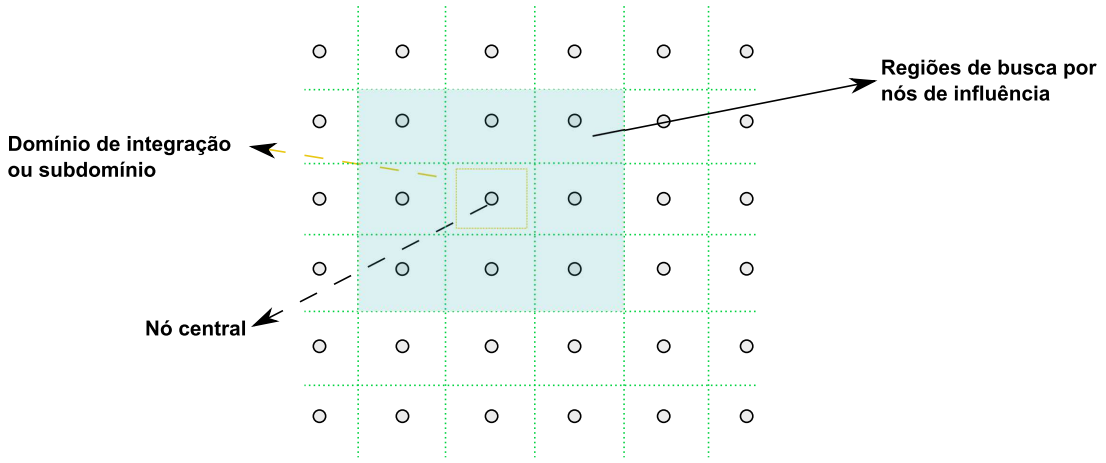


Figura 4.12: Ilustração mostra como o domínio do problema pode ser subdividido em regiões, para se evitar a concorrência do acesso à memória global na GPU.

Nas etapas 7 e 11, a diferença na implementação é a mesma, ao se aproximar os valores de ϕ e $\frac{d\phi}{dn}$ deve-se utilizar o MLS. Apenas a título de comparação posterior, o MLS tem ordem de complexidade $O(m^2)$ determinada pela inversão da matriz A da equação 3.46, onde m é o número de monômios do polinômio base.

Pode-se destacar também que no MLS, como visto no capítulo anterior, possui um termo a mais na equação da forma fraca, para forçar a condição de contorno essencial. Isto pode ser notado na equação 3.95, repetida a seguir por conveniência.

$$\int_{\Gamma_s u} \hat{n} \cdot \nabla u d\Gamma + \int_{\Gamma_s q} \hat{n} \cdot \nabla u d\Gamma + \int_{\Gamma_s u} \hat{n} \cdot \nabla u d\Gamma + \int_{\Gamma_s u} \alpha(u - \tilde{u}) d\Gamma - \int_{\Omega_s} p d\Omega = 0 \quad (4.1)$$

Este termo irá influenciar na etapa oito, onde se avalia a forma fraca, uma vez que, o termo da penalidade será integrado numericamente, e contribuirá na matriz do sistema de equações global. Este termo é copiado na equação 4.2 separadamente.

$$\int_{\Gamma_{su}} \alpha(u - \tilde{u}) d\Gamma \tag{4.2}$$

A adição deste termo da penalidade na avaliação da forma fraca local influencia diretamente no tempo de execução na GPU, bem como na precisão da solução obtida. Esses efeitos serão estudados no próximo capítulo.

4.6.4 MLPG+RPIMp

O MLPG com RPIMp (do inglês *radial point interpolation method with polinomial reproduction*) tem certas vantagens em relação ao MLPG+MLS, em termos de facilidade de implementação na GPU.

Na etapa três da figura 4.7, o MLPG+RPIMp não é necessário determinar quais nós influenciam os pontos de integração no subdomínio para montar o domínio de suporte. O MLPG+RPIMp precisa somente de uma aproximação local da função de forma [13], sendo assim, só necessário determinar quantos nós farão parte do domínio de suporte de um subdomínio, e criar um algoritmo para obter esses nós de maneira que o domínio de suporte não fique desbalanceado (com nós somente de um lado, ou nós mal distribuídos). O número de nós que fazem parte do domínio de suporte no MLPG+RPIMp é um parâmetro do método.

A figura 4.13 ilustra a busca pelo nós no domínio de suporte do RPIMp. A busca começa com um determinado raio $R1$. Todos os nós à distância $R1$ do nó central são candidatos para o domínio de suporte. Caso não se atinja o número desejado de nós, o raio é incrementado de um valor constante, para incluir mais nós, e a busca continua até que se atinja o número desejado de nós.

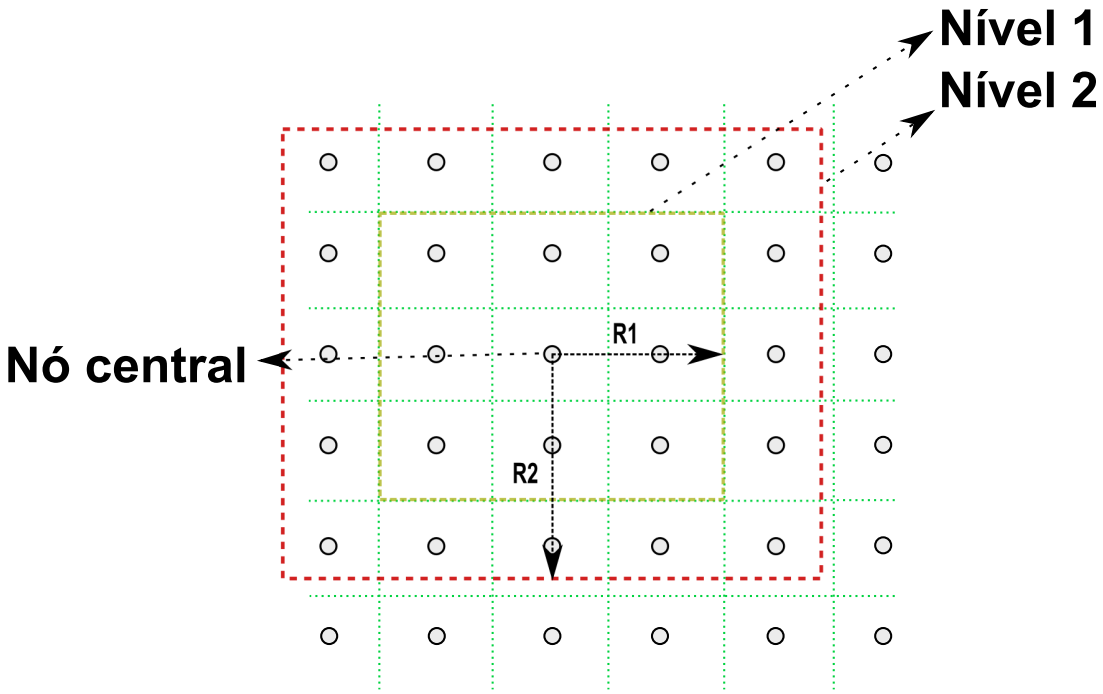


Figura 4.13: A busca pelo domínio de suporte no RPIMp é feita incrementando-se um raio R em (R1,R2,...), até se obter o número desejado de nós. A busca começa pelos nós dentro de um raio R1 do nó central, caso o número de nós dentro dessa região não atingir o valor desejado, o raio é incrementado e uma nova busca acontece, até se atingir o número desejado no domínio de suporte.

A mudança nas etapas sete e onze na figura 4.7 é evidente, dado que o método de interpolação é diferente. Porém, mais uma vez, a título de comparação posterior, nota-se que no RPIMp a operação mais cara é a inversão da matriz R na equação 3.87 e a inversão na equação 3.85, repetidas nas equações 4.3 e 4.4.

$$\mathbf{S}_a = \mathbf{R}_Q^{-1}[\mathbf{1} - \mathbf{P}_m\mathbf{S}_b] = \mathbf{R}_Q^{-1} - \mathbf{R}_Q^{-1}\mathbf{P}_m\mathbf{S}_b \tag{4.3}$$

$$S_b = (\mathbf{P}_m^T\mathbf{R}_Q^{-1}\mathbf{P}_m)^{-1}\mathbf{P}_m^T\mathbf{R}_Q^{-1} \tag{4.4}$$

A partir das equações do RPIMP, pode-se dizer que o método tem que inverter a matriz R e a matriz $(\mathbf{P}_m^T \mathbf{R}_Q^{-1} \mathbf{P}_m)^{-1}$. A primeira é uma matriz nxn onde n é o número de nós no domínio de suporte com complexidade $O(n^2)$, e a segunda é uma matriz mxm, onde m é o tamanho do vetor de polinômios. Neste trabalho m é igual 3, dado que se utiliza uma base linear.

Finalmente, a etapa oito, onde se impõem as condições de contorno essenciais, é totalmente diferente do MLS. Nesta etapa, o algoritmo simplesmente fixa o valor da função no nó, com o respectivo valor da condição de contorno. Para isto, basta tomar a linha da matriz K, correspondente ao subdomínio que está sendo avaliado, e fixar o valor 1 na diagonal e zero no restante, enquanto, na mesma linha no vetor F, fixa-se o valor da condição de contorno. Sendo assim, a *thread* cujo nó está na fronteira de Dirichlet irá simplesmente testar se o nó está na fronteira, e fixar os valores diretamente no sistema de equações.

Capítulo 5

Resultados

Após descrever a teoria e implementação do algoritmo MLPG na GPU, nesta seção são apresentados os resultados, mostrando a comparação das duas versões do método MLPG apresentadas (MLPG+MLS e MLPG+RPIMp). A solução também é comparada com a solução analítica do problema, e o erro médio quadrático é levantado para todas as simulações.

Como o objetivo desta dissertação não é mostrar que o MLPG funciona bem para os problemas eletromagnéticos (isso foi feito em [9]), e sim verificar o desempenho de sua implementação na GPU, soluciona-se um problema físico suficientemente simples mas que proporcione resultados para uma boa discussão e que possua solução analítica. O problema adotado, do capacitor, é ilustrado na figura 5.1:

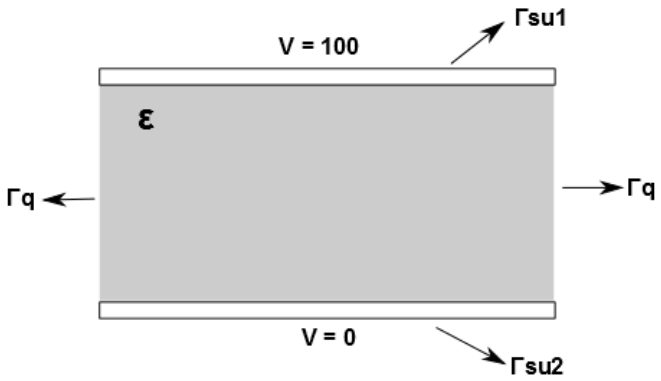


Figura 5.1: Capacitor de placas paralelas: geometria explicitando domínio e fronteira.

A solução da equação diferencial irá determinar o valor do potencial em cada ponto, considerando um material dielétrico com permissividade elétrica constante. A equação diferencial governante (equação 5.1) é obtida a partir da lei de Gauss, considerando um meio livre de cargas.

$$\nabla^2 \Phi = 0 \tag{5.1}$$

Submetida às condições de contorno:

$$\Phi = 1V, \ \ em \ \ \Gamma_{su1}, \tag{5.2a}$$
$$\Phi = 0V, \ \ em \ \ \Gamma_{su2}, \tag{5.2b}$$
$$\frac{\partial \Phi}{\partial \hat{n}} = 0, \ \ em \ \ \Gamma_q \tag{5.2c}$$

Onde Γ_{su1} e Γ_{su2} são as fronteira de Dirichlet, onde são submetidas as condições essenciais de contorno. Γ_q é a fronteira de Neumann, onde é submetida a condição de contorno natural.

O método MLPG possui diversos parâmetros, que foram citados nos capítulos anteriores, e cujos valores, para as simulações a seguir, são apresentados na tabela 5.1.

É importante ressaltar também a necessidade da GPU suportar operações com ponto flutuante de precisão dupla. Isso se deve ao fato do algoritmo não produzir resultado algum com precisão simples. Essa é uma característica da própria formulação do método MLPG, uma vez que a matriz global produzida pelo MLPG possui um número de condição elevado o que inviabiliza o uso da representação de ponto flutuante de precisão simples. Portanto, deve-se utilizar uma placa de vídeo com capacidade 3.0 ou superior (a NVIDIA classifica as placas de vídeo com um número que representa a sua capacidade [5]), caso se utilize uma GPU da NVIDIA.

Na tabela 5.2 são apresentados os valores numéricos das simulações. Esses valores aparecem resumidos agora, porém serão mostrados na forma de gráficos nas seções que se seguem.

Tabela 5.1: Parâmetros utilizados na implementação do MLPG.

Parâmetro	Descrição	Valor
DIM (Parâmetro da GPU)	Número de nós em uma dimensão	16 até 352
DIMT (Parâmetro da GPU)	Número de threads por bloco para a chamada do kernel	16
Rinfluencia	raio de influencia para as funções de base radial do MLS	1.45
Rintegração	raio de integração ou raio do subdomínio	0.5
poli_base	Número de monômios do polinômio base do MLS.	3
$n\_neighbors$	Número de vizinhos considerados para avaliação no domínio de suporte do MLS	9
$alpha\_mls$	valor de α, parâmetro do método da penalidade	$1x10^3$ até $1x10^6$
NInfluenceNodes	número de nós no domínio de influência no RPIMp	9
NpontosIntegracao	número de pontos de integração utilizados	3

A primeira coluna da tabela é o número de nós em uma dimensão. Como o domínio do problema é quadrado, o número total de nós é o quadrado deste valor. Chega-se então no valor da segunda coluna, o total de nós no domínio. As próximas colunas apresentam duas grandes divisões, os teste com o MLPG+MLS e os testes com o MLPG+RPIMp. Nas duas primeiras sub-colunas (GPU e CPU) estão os repectivos tempos de execução da montagem da matriz global do MLPG, respectivamente na GPU e na CPU em ms. As colunas seguintes apresentam o número de condição da matriz e, por último, o erro médio quadrático.

O erro quadrático médio foi calculado utilizando-se a equação 5.3 a seguir.

$$Erro = \left(\frac{\sum_{i=1}^{n} (\Phi_{exato} - \Phi_{calculado})^2}{\text{Número de nós}} \right)^{\frac{1}{2}} \tag{5.3}$$

Tabela 5.2: Comparação entre os resultados da montagem do sistema de equações na GPU e na CPU do MLPG+MLS e MLPG+RPIMp. Os tempos estão expressos em ms

Tamanho do domínio	Número de nós	MLS					RPIMp				
		GPU [ms]	CPU [ms]	Número de condição	Erro	SpeedUp (tempo cpu/tempo gpu)	GPU [ms]	CPU [ms]	Número de condição	Erro	SpeedUp
16	256	2,2	2	2,07E+04	7,41E-15	9,09E-01	2,2	2	1,70E+08	1,09E-11	9,09E-01
32	1024	2,2	4,7	4,09E+04	6,30E-15	2,14E+00	2,4	10	2,16E+09	2,71E-11	4,17E+00
48	2304	2,5	8	6,10E+04	9,79E-15	3,20E+00	2,7	22	1,72E+10	9,16E-11	8,15E+00
64	4096	2,7	12,1	8,13E+04	9,31E-15	4,48E+00	3,2	39	7,16E+10	3,35E-10	1,22E+01
80	6400	3,6	16,7	1,02E+05	5,92E-15	4,64E+00	4,4	62	2,20E+11	5,02E-10	1,41E+01
96	9216	5,4	22,2	1,22E+05	4,64E-15	4,11E+00	7,1	89	5,48E+11	3,53E-09	1,25E+01
112	12544	6,4	28,4	1,42E+05	5,01E-15	4,44E+00	7,8	121	1,19E+12	8,15E-09	1,55E+01
128	16384	8,1	35,2	1,63E+05	5,59E-15	4,35E+00	9,2	162	2,33E+12	4,71E-09	1,76E+01
144	20736	9,7	43,2	1,83E+05	9,20E-15	4,45E+00	11,9	204	4,21E+12	5,76E-09	1,71E+01
160	25600	11,8	50,7	2,04E+05	7,15E-15	4,30E+00	14,9	252	7,15E+12	2,15E-08	1,69E+01
176	30976	13,6	60	2,25E+05	5,71E-15	4,41E+00	17	309	1,15E+13	2,25E-08	1,82E+01
192	36864	16,4	69,4	2,45E+05	1,28E-14	4,23E+00	20,1	362	1,78E+13	4,17E-08	1,80E+01
208	43264	19	80	2,66E+05	7,51E-15	4,21E+00	23,4	429	2,67E+13	1,29E-07	1,83E+01
224	50176	21,7	91,2	2,87E+05	8,71E-15	4,20E+00	26,4	493	3,87E+13	4,24E-08	1,87E+01
240	57600	24,4	103,2	3,08E+05	7,75E-15	4,23E+00	29,6	573	5,46E+13	1,70E-07	1,94E+01
256	65536	27,7	114,4	3,29E+05	6,63E-15	4,13E+00	33,7	642	7,55E+13	3,38E-07	1,91E+01
272	73984	30,6	127,5	3,50E+05	1,02E-14	4,17E+00	37,7	729	1,02E+14	4,67E-08	1,93E+01
288	82944	34,1	141,3	3,71E+05	1,36E-14	4,14E+00	42,4	814	1,36E+14	4,49E-08	1,92E+01
304	92416	37,7	155,1	3,92E+05	5,22E-15	4,11E+00	46,2	905	1,79E+14	2,53E-07	1,96E+01
320	102400	41,5	171,8	4,14E+05	1,40E-14	4,14E+00	51	1006	2,31E+14	1,10E-07	1,97E+01
336	112896	45,9	186,4	4,35E+05	2,21E-14	4,06E+00	55,6	1113	2,95E+14	5,05E-07	2,00E+01
352	123904	50	202,5	4,56E+05	7,51E-15	4,05E+00	60,9	1218	3,73E+14	4,64E-07	2,00E+01

Para avaliação do erro quadrático médio foi necessário resolver o sistema de equações do MLPG. O método utilizado para se resolver o sistema de equações neste trabalho foi a função *mldivide* do MATLAB versão 7.12 R2011a. O algoritmo implementado por essa função é capaz de identificar o melhor método computacional para resolver o sistema, a partir de uma análise da matriz [6]. Este método é uma variação do método LU iterativo recursivo, que computa a decomposição LU de uma matriz M por N qualquer utilizando pivotação parcial com intercâmbios de linhas [4]. Os dados calculados pelo algoritmo em C++ foram passados para o *script* em MATLAB por arquivos de texto. Isso foi feito preocupando-se com o número de casas decimais nos arquivos (20 casas após a vírgula), para que a precisão dos dados não fosse perdida.

Na tabela 5.3 são mostradas as configurações de *hardware* utilizadas neste trabalho. O ambiente de *software* escolhido foi a IDE (do inglês *interface development environment*) Microsoft Visual Studio 2010 com Parallel Nsight para Visual Studio versão 3.1, e *driver* para CUDA versão 5.5.

Tabela 5.3: Configuração da CPU e da GPU utilizadas neste trabalho.

CPU	Intel(R) Core i7-3770 CPU @ 3.40GHs, 8GB de memória RAM, e sistema operacional Windows 7 64bits
GPU	NVIDIA GeForce GTX 680, com capacidade CUDA 3.0, 2GB de memória global, com 1536 núcleos CUDA

5.1 Análise dos resultados da montagem do sistema de matrizes na GPU e na CPU do MLPG+MLS

Com o intuito de verificar se a paralelização do método MLPG+MLS foi bem sucedida, os tempos de execução na CPU (implementação serial) e na GPU (implementação paralela) são comparados no gráfico da figura 5.2. Neste gráfico estão presentes o tempo de execução em milisegundos *versus* o número de nós.

É possível notar que a GPU tem uma ampla vantagem sobre a CPU, executando até 4 vezes mais rápido que o mesmo algoritmo rodando na CPU. E à medida em que se aumenta o número de nós no domínio, essa diferença se torna cada vez maior.

Não está evidente no gráfico, mas é possível concluir, a partir da primeira linha da tabela da figura 5.2, que existe um cruzamento entre as retas. Este cruzamento é justificável pelo fato do algoritmo na GPU possuir um *overhead* de transferência de dados entre a memória da CPU e a memória da GPU, e, por isso, quando a quantidade de nós é pequena, o tempo de transferência se torna maior que o tempo da montagem do sistema. Portanto, somente para domínios com poucos nós a CPU tem vantagem sobre a GPU.

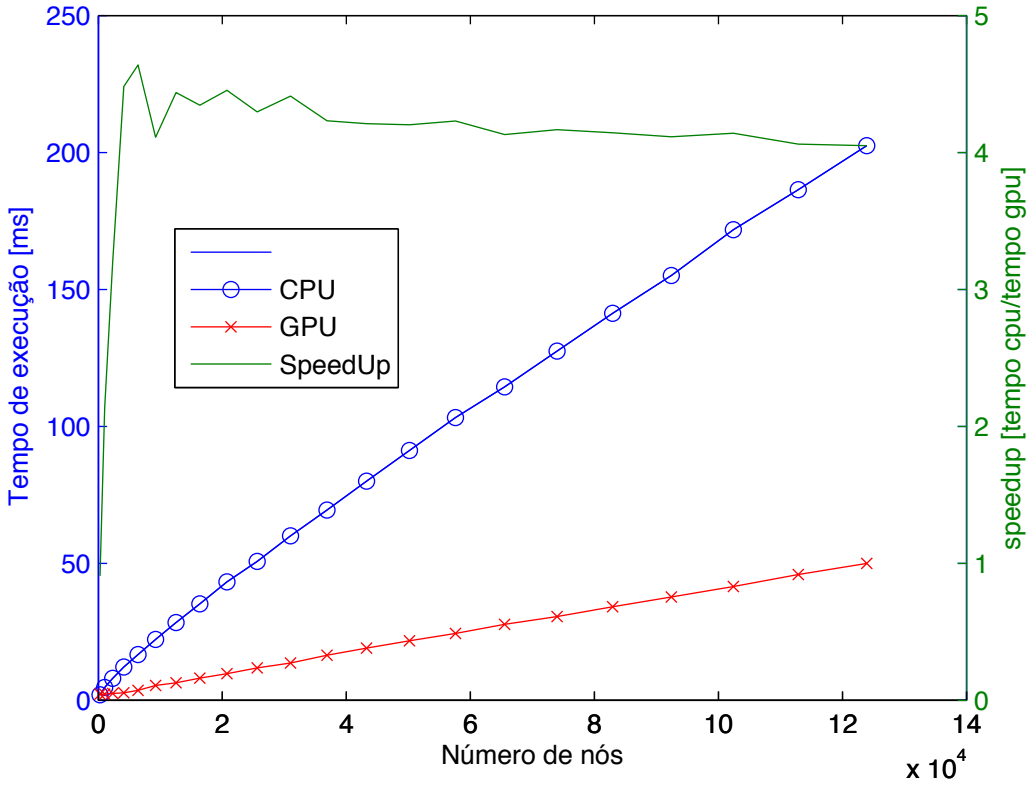


Figura 5.2: Comparação entre os tempos de execução na GPU e na CPU no MLPG+MLS aumentado-se o número de nós no domínio do problema.

5.2 Análise dos resultados da montagem do sistema de matrizes na GPU e na CPU do MLPG+RPIMp

O gráfico da figura 5.3 mostra os mesmos resultados de tempo de execução na GPU e na CPU, porém para o caso MLPG+RPIMp. Nota-se também um ganho bastante expressivo com o algoritmo paralelo, até 20 vezes mais rápido, com $352x352 = 123904$ nós.

A intenção em utilizar tanto o RPIMp quanto o MLS para gerar as funções de forma é justamente comparar os dois algoritmos quanto ao tempo de execução. Como foi observado no capítulo anterior, a operação mais cara do MLPG+MLS é a inversão de uma matriz mXm, onde m é o número de monômios do polinômio base do MLS. Já no MLPG+RPIMp a operação mais cara é a inversão de duas matrizes nXn e mXm, n é o número de nós do domínio de suporte enquanto m é o número de monômios da base polinomial, que para os testes em

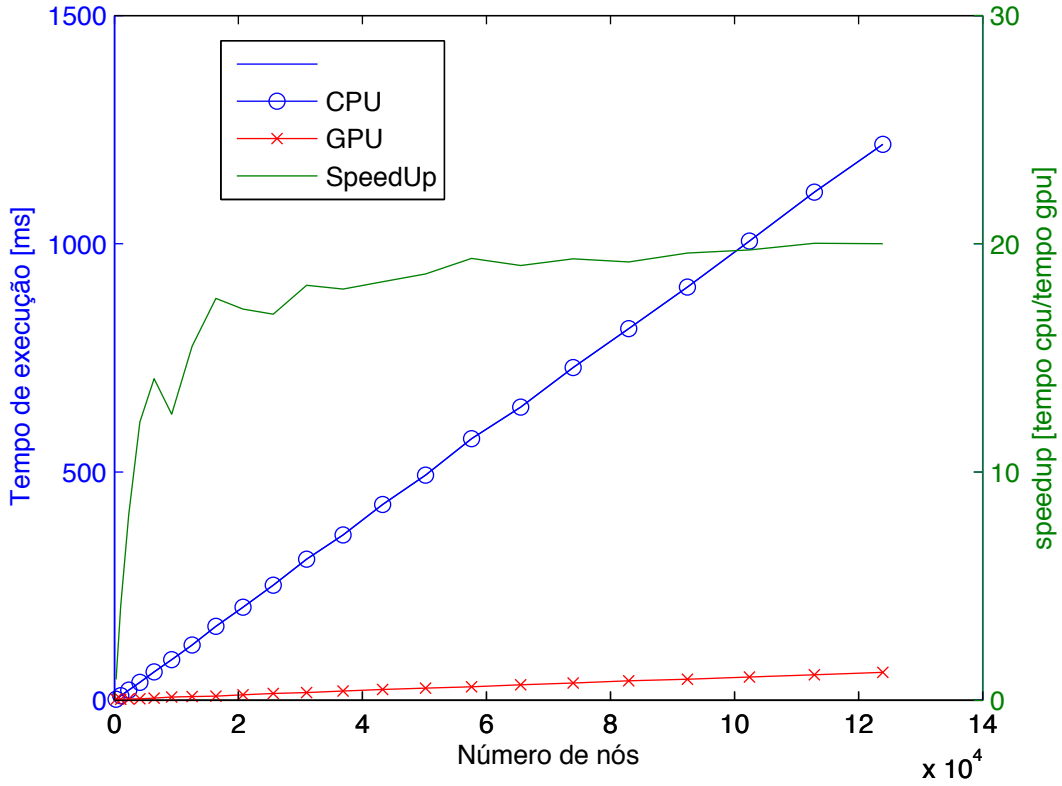


Figura 5.3: Comparação do tempo de execução na GPU e na CPU em ms aumentado-se o número de nós no domínio do problema, para MLPG+RPIMp.

questão m é igual a 3 (polinômio de grau um). O número de nós do domínio de suporte no MLS e no RPIMp é igual. Como pode ser notado na tabela 5.1, no MLS $n\_neighbors = 9$ e no RPIMp $NInfluenceNodes = 9$ também. Ou seja, para o MLS tem-se que inverter uma matriz $3x3$ e realizar operações de multiplicação e soma de matrizes de tamanho até $9x9$, enquanto no RPIMp se tem duas matrizes uma de $9x9$ e a outra $3x3$ para inverter e multiplicações e somas de matrizes de até $9x9$. Portanto é de se esperar que o MLS tenha tempos de execução menores que o RPIMp, como pode ser visto na figura 5.4, que compara os tempos de execução na GPU do MLPG+MLS e MLPG+RPIMp com o número de nós no domínio de suporte igual em ambos os métodos.

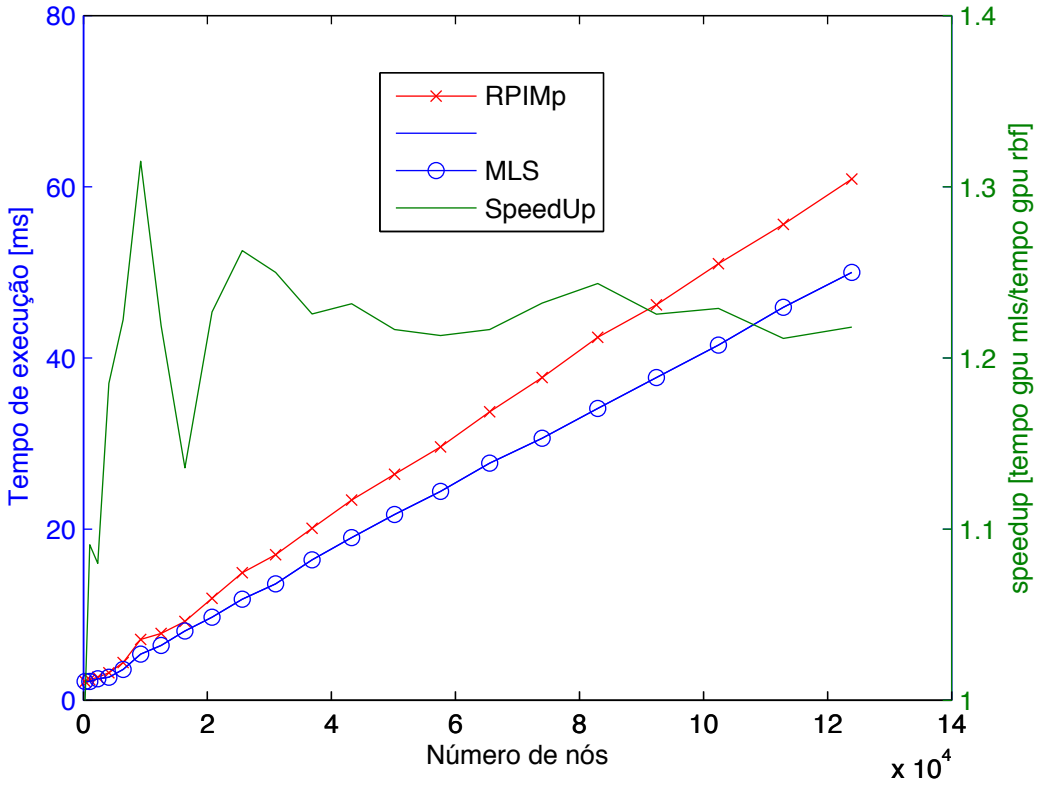


Figura 5.4: Comparação do tempo de execução na GPU entre os métodos MLPG+MLS e MLPG+RPIMp.

Neste gráfico pode-se notar que o RPIMp é até $1,2$ vezes mais lento que o MLS.

5.3 Problema do mal condicionamento da matriz do sistema global e comparação entre os métodos utilizando RPIMp e MLS

Existe um problema bem conhecido do método MLPG que é o mal condicionamento do sistema global de equações. O número de condição da matriz do sistema fornece uma medida da exatidão dos resultados da solução do sistema de equações lineares global.

O número de condição da matriz do sistema é definido por:

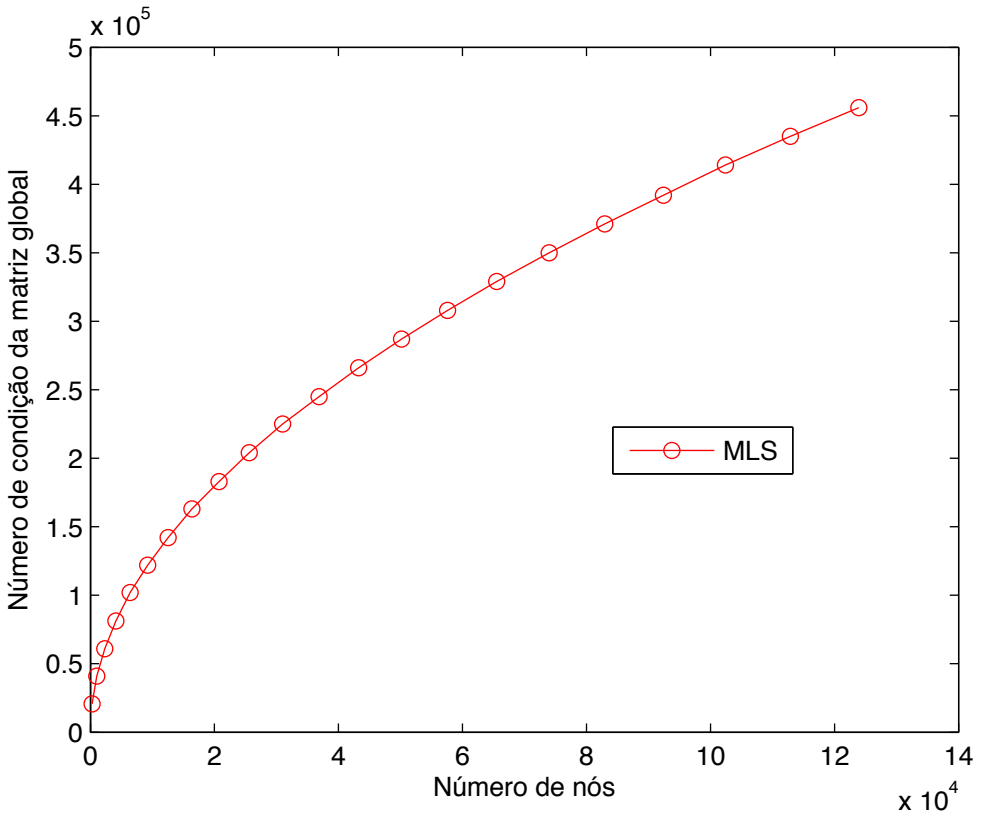


Figura 5.5: Variação do número de condição da matriz global para o método MLPG+MLS à medida em que se aumenta o número de nós no domínio.

$$Cond = max(K)/min(K) \tag{5.4}$$

Onde $max(K)$ e $min(K)$ são o maior e o menor autovalor absoluto da matriz K, respectivamente. Quanto maior o número de condição da matriz pior é a exatidão dos resultados. Analogamente, quanto mais próximo de 1 o número de condição, mais precisos são os resultados.

Nas figura 5.5 e 5.6 apresenta-se o comportamento do número de condição à medida em se aumenta o número de nós no domínio para o MLPG+MLS (figura 5.5) e para o MLPG+RPIMp (figura 5.6).

Em ambos os gráficos, tanto no caso do MLS quanto no caso do RPIMp, há um aumento do número de condição da matriz do sistema quando se aumenta o número de nós no domínio do

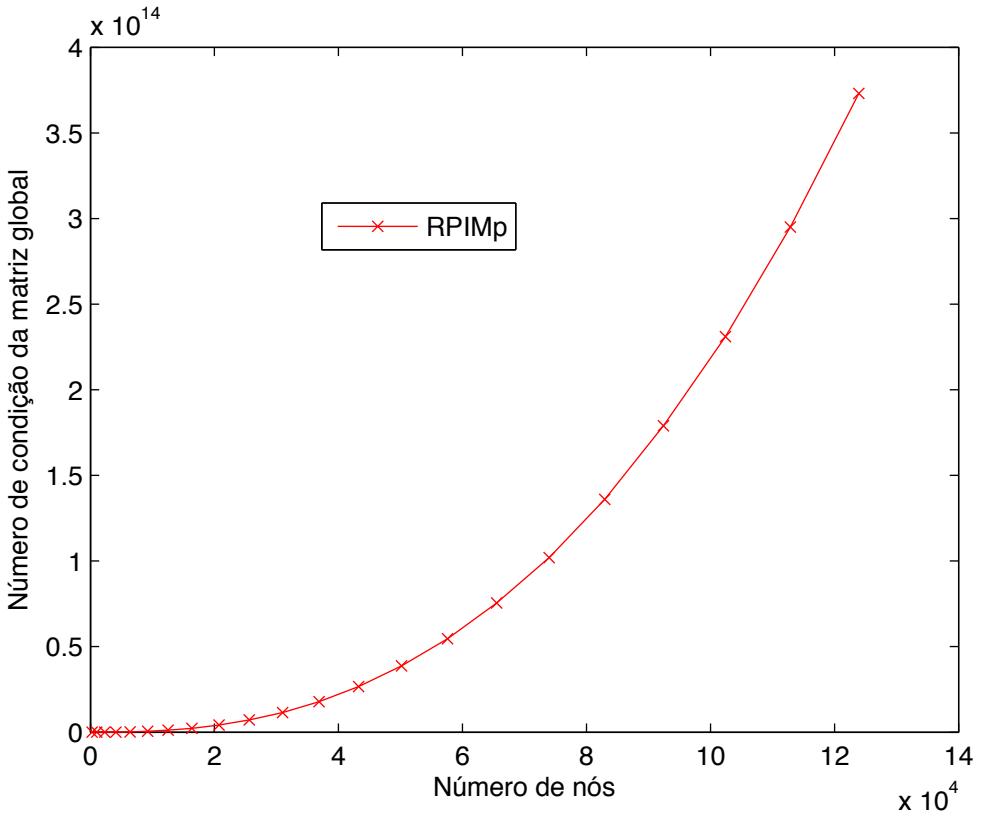


Figura 5.6: Variação do número de condição da matriz global para o método MLPG+RPIMp à medida em que se aumenta o número de nós no domínio.

problema. Com o aumento do número de condição ocorre também uma redução na precisão da solução do sistema de equações lineares gerado e, no caso do RPIMp,não se obteve redução no erro da solução com o aumento do número de nós, como pode ser visto na tabela 5.2.

5.4 Estratégia para melhorar o condicionamento da matriz no MLPG+MLS

A piora do condicionamento da matriz global é um ponto a melhorar ainda no método MLPG em geral. Nesta seção apresenta-se uma forma de se tentar reduzir o número de condição para o caso MLPG+MLS.

O parâmetro de penalidade no método MLS possui normalmente um valor muito alto, se comparado com os valores avaliados dos outros termos da forma fraca local. Portanto, na

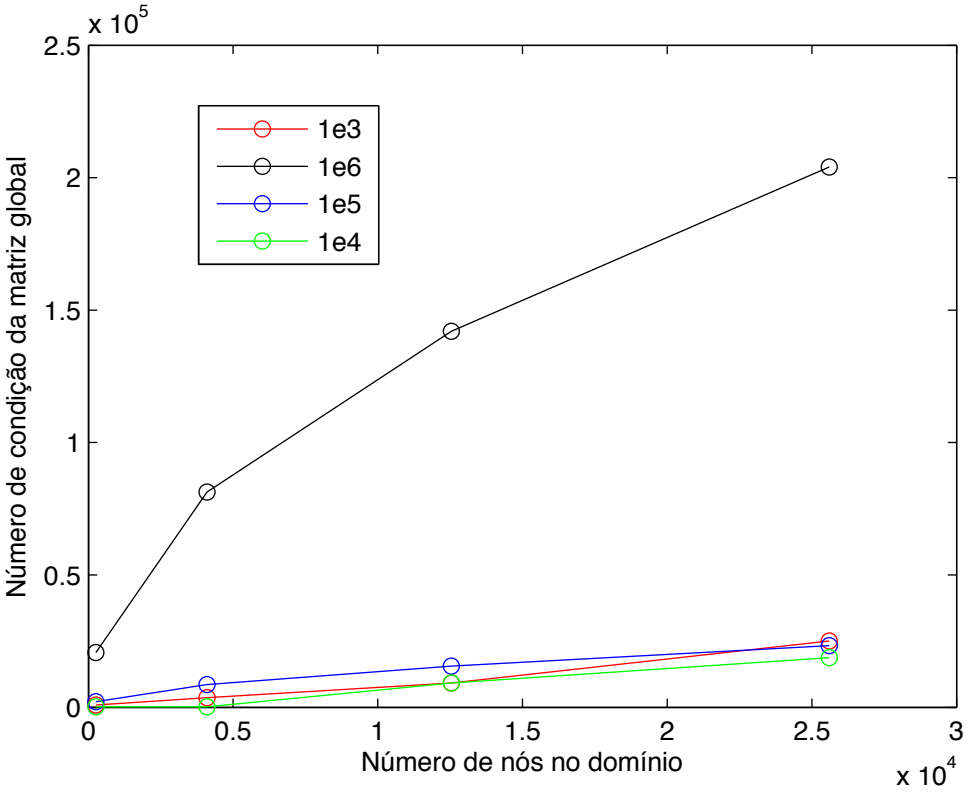


Figura 5.7: Número de condição da matriz global (MLPG+MLS) para diferentes valores de penalidade em função do número de nós utilizados.

figura 5.7 é mostrado um gráfico do número de condição da matriz em função do número de nós no domínio, para quatro valores de penalidade ($1x10^6$, $1x10^5$, $1x10^4$, $1x10^3$).

É possível identificar uma redução no número de condição da matriz, quando se reduz este parâmetro. A desvantagem de se fazer este tipo de ajuste é que o parâmetro de penalidade varia quando se muda o problema físico avaliado. Portanto, para cada problema que se deseja obter uma solução deve-se ajustar o parâmetro de penalidade, e este valor é determinado através de várias experimentações.

A tabela 5.4 exibe os valores utilizados para se construir a figura 5.7. Nela pode-se notar que no melhor dos casos o número de condição teve uma redução de $2,07x10^4$ para $2,38x10^2$, e que no pior dos caso essa melhoria foi de $2,04x10^5$ para $1,8x10^4$.

Tabela 5.4: Dados da simulação da tentativa de reduzir o número de condição da matriz global do MLS.

Tamanho do domínio	Número de nós	Penalidade	MLS		
			Solução do Sistema [ms]	Número de condição	Erro
16	256	1,00E+06	1,798	5,17E+05	1,73E+01
64	4096	1,00E+06	2,163	2,77E+06	0,284
112	12544	1,00E+06	3,149	5,75E+06	0,0425
160	25600	1,00E+06	5,78	9,47E+06	0,0171
16	256	1,00E+05	1,74	5,20E+04	1,74E+00
64	4096	1,00E+05	2,162	2,84E+05	2,91E-02
112	12544	1,00E+05	3,083	6,01E+05	4,50E-03
160	25600	1,00E+05	6,04	1,01E+06	1,90E-03
16	256	1,00E+04	1,784	5,52E+03	1,82E-01
64	4096	1,00E+04	2,162	8,23E+04	3,80E-03
112	12544	1,00E+04	3,145	3,01E+05	8,31E-04
160	25600	1,00E+04	4,711	7,08E+05	3,78E-04
16	256	1,00E+03	1,191	3,67E+03	3,47E-02
64	4096	1,00E+03	2,158	8,23E+04	1,70E-03
112	12544	1,00E+03	4,084	3,00E+05	5,42E-04
160	25600	1,00E+03	4,707	7,08E+05	2,64E-04

Capítulo 6

Conclusão

Neste trabalho apresentou-se uma forma de paralelização do método sem malha Petrov-Galerkin na GPU. O método MLPG é uma solução inovadora para se resolver problemas de engenharia, principalmente para problemas com grandes deformações ou descontinuidades. O método elimina a necessidade de remalhar o domínio do problema e se mostra como um verdadeiro método sem malha. Existem diversos problemas que necessitam de métodos numéricos com diferentes abordagens e o MLPG se mostra uma opção atrativa.

Inicialmente, apresentaram-se os conceitos básicos dos métodos sem malha e efetuou-se uma comparação com o tradicional método dos elementos finitos. A diferença na discretização do domínio no FEM e nos métodos sem malha foi ilustrada. Mostrou-se em seguida o método MLPG definido em linhas gerais, juntamente com a definição de dominio de suporte e domínio de influência. Apresentar-se também um resumo da teoria eletromagnética, com ênfase na eletrostática e magnetostática, símbolos e definições utilizadas.

O MLPG foi adaptado para a GPU por ser baseado em uma forma local fraca e pelo fato principal de que cada avaliação dessa forma local fraca não depende das demais e, consequentemente, o método não necessita de nenhuma sincronização. O MLPG admite várias escolhas de funções de forma e de teste, neste trabalho foram apresentadas mas neste trabalho utilizou-se o MLPG5 porque a função de Heaviside utilizada no versão como função de teste, o torna bastante simples e interessante para paralelização, pois elimina o termo da integração de domínio na forma fraca local.

Para a geração das funções de forma, foram escolhidos dois métodos, o MLS e o RPIMp. Assim, foi possível obter um comparativo do comportamento dos dois métodos na GPU. O MLS se mostra bastante simples e rápido se comparado com o RPIMp com o mesmo número de nós no domínio de suporte. Porém, no MLS é necessária a utilização de técnicas adicionais para se impor as condições de contorno essenciais e, para muitos problemas, pode ser uma tarefa difícil determinar o parâmetro α de penalidade usado para impor estas condições de contorno.

Por outro lado, no RPIMp não há a preocupação com raios de influência na determinação dos nós do domínio de suporte, como ocorre no MLS. A maneira como são determinados os nós no domínio de suporte do RPIMp é simplificada. No RPIMp, somente é necessária uma aproximação local ao redor de um ponto, portanto pode-se escolher um determinado número de nós bem distribuídos na vizinhança do ponto de interesse, sem se preocupar com raios de influência dos nós. Isso facilita bastante a adaptação do método para funcionar com distribuições nodais aleatórias na GPU. Por outro lado, o MLS precisa de soluções mais elaboradas no caso de distribuições nodais aleatórias para se determinar os nós presentes no domínio de suporte de um ponto. Neste trabalho, explorou-se somente um problema com distribuição nodal uniforme, porém para o caso mais complexo com diferentes densidades de nós no domínio do problema, o RPIMp teria uma vantagem significativa em relação ao MLS.

No capítulo quatro apresentou-se um estudo sobre a evolução das arquiteturas das GPU's ao longo da última década. Concentrou-se, principalmente, nas arquiteturas da NVIDIA, por incorporar o formato de programação CUDA, que permite que aplicações para a GPU sejam programadas em C de maneira fácil e direta. A arquitetura Tesla, a pioneira na programação de propósito geral em GPU's, foi estudada em detalhes, pelo fato de proporcionar um estudo dos principais conceitos do hardware capazes de tornar o pipeline programável. As mesmas estruturas presentes na arquitetura Tesla estão até hoje presentes nas arquiteturas mais novas, a não ser por diferenças no tamanho de cache, otimização do consumo de energia, melhoramentos gerais nos gráficos e um aumento na memória de *cache* e memória global.

Para validar a implementação do método na GPU e, efetivamente, medir o ganho real obtido em relação à implementação na CPU, foram realizados diversos testes para medir o tempo de execução da aplicação na GPU e na CPU. Para isso resolveu-se um problema eletrostático simples, o capacitor de placas paralelas, cuja solução analítica é obtida facilmente. Os resultados foram validados comparando-se a solução obtida com a solução analítica, e constatou-se que

os erros obtidos na CPU e na GPU são iguais para cada valor de número de nós no domínio considerando um determinado método (MLPG+MLS ou MLPG+RPIMp). As duas versões do MLPG, MLPG+MLS e MLPG+RPIMp, foram comparadas quanto ao tempo de execução da montagem do sistema e quanto ao número de condição da matriz do sistema de equações. Os resultados mostraram boa precisão com relação à solução analítica, além de um ganho de até 20 vezes no tempo de execução na montagem da matriz. Avaliou-se também o aumento do número de condição da matriz do sistema de equações à medida em que se aumenta o número de nós. O sistema de equações não é um sistema fácil de se resolver pelos altos valores de número de condição, comumente obtidos com o MLPG. Não se utilizou métodos de solução de sistemas de equações em paralelo neste trabalho. Existem diversos métodos ainda em desenvolvimento para solucionar tais sistemas, e eles serão objeto de investigações futuras.

6.1 Trabalhos futuros e melhorias

Um ponto de partida para trabalho futuro seria a paralelização da solução do sistema de equações do MLPG. Utilizando-se métodos iterativos ou não, esse é um grande desafio devido ao mal-condicionamento das matrizes geradas pelo método. Nesse sentido, se torna extremamente interessante o desenvolvimento de algoritmos paralelos na GPU capazes de resolver o sistema de maneira satisfatória. Existem diversas bibliotecas de solução de sistemas lineares em desenvolvimento, que fornecem métodos paralelizados na GPU. Dentre elas, pode-se citar o projeto de código aberto da CUSP [3].

Outro melhoramento seria na busca espacial por nós no domínio de suporte na GPU. Isso viabilizaria o uso do MLPG+MLS na GPU, considerando distribuições nodais não uniformes, o que não foi feito neste trabalho.

Há também a possibilidade de resolver o sistema de equações sem mesmo tê-lo por completo em memória. Esta abordagem já existe e é conhecida como EbE (Element-by-Element), algo já implementado e testado para o método dos elementos finitos na GPU [10]. Esta formulação necessita ainda do desenvolvimento de métodos paralelos iterativos capazes de resolver o sistema de equações do MLPG na GPU, incluindo o desenvolvimento de pré-condicionadores que consigam diminuir o número de condição da matriz do sistema de maneira satisfatória.

Uma última melhoria seria a integração dos códigos do MLPG na GPU com o *framework* desenvolvido pelo aluno Naisses Zoia Lima, durante o seu mestrado na Universidade Federal

de Minas Gerais (UFMG). Este *framework* proporciona uma alta flexibilidade e reuso do código, além de constituir uma ferramenta unificada de métodos sem malha para os alunos que continuarão o desenvolvimento desses algoritmos [12].

Referências Bibliográficas

[1] GeForce-GTX-680-WhitePaper. http://www.geforce.com/Active/en_US/en_US/pdf/GeForce-GTX-680-Whitepaper-FINAL.pdf, 2012.

[2] How to Access Global Memory Efficiently in CUDA C/C++ Kernels. https://developer.nvidia.com/content/how-access-global-memory-efficiently-cuda-cc-kernels, janeiro 2013.

[3] CUSP library. https://code.google.com/p/cusp-library/, 2013.

[4] DGETRF Lapack. http://www.netlib.org/lapack/explore-html/d0/d39/_v_a_r_i_a_n_t_s_2lu_2_r_e_c_2dgetrf_8f.html.

[5] CUDA C programming guide. http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf, july 2013.

[6] mldivide Left or Right Matrix Division. http://www.maths.lth.se/na/courses/NUM115/NUM115-11/backslash.html.

[7] Satya N. Atluri e Shengping Shen. The Meshless Local Petrov-Galerkin (MLPG) method: A simple and Less-Costly alternative to finite element and boundary element methods. CMES, vol. 3, no. 1, pp.11-51, 2002.

[8] Stuart Oberman John Montrym Erik Lindholm, John Nickols. NVIDIA Tesla: A unified graphics and computing architecture. IEEE Computer Society, 2008.

[9] Alexandre Ramos Fonseca. Algoritmos Eficientes em Metodos Sem Malha. tese de doutoramento, Universidade Federal de Minas Gerais, 2011.

[10] Zsolt Badics Imre Kiss, Szabolcs Gyimothy e Jozsef Pavo. Parallel realisation of the element-by-element FEM technique by CUDA. IEEE Transactions on magnetics, 2012.

[11] Wendy Jones. Beginning DirectX 10 Game Programming By Wendy Jones. Thomson Course Technology PTR, 1st edição, 2007.

[12] Naisses Zoia Lima. Desenvolvimento de um Framework para Metodos Sem Malha. tese de doutoramento, Universidade Federal de Minas Gerais, 2011.

[13] G. R. Liu. Mesh Free methods: Moving beyond the finite element method. CRC Press, 1st edição, 2010.

[14] Dawn R. Phillips. Meshless Local Petrov-Galerkin Method for Bending Problems. tese de doutoramento, Langley Research Center, Hampton, Virginia, 2002.

[15] Bill Licea-Kane Randi J. Rost. OpenGL shading language. John Wiley and Sons, 3rd edição, 2009.

[16] T. Zhu S. N. Atluri. A new meshless Local Petrov-Galekin (MLPG) approach in computational mechanics. Computational mechanics, Vol. 22, páginas 117–127, 1998.

[17] Theodore Van Duzer Simon Ramo, John R. Whinnery. Fields and waves in communication electronics. John Wiley and Sons, 3rd edição, 1994.

[18] L. Xuan e L. Upda. Element free Galerkin method for static and quasi-static electromagnetic field computation. IEEE Transactions on Magnetics, 2004.